Méthodologie de conception pour la virtualisation et le déploiement d'applications parallèles sur plateforme reconfigurable matériellement

Clément Foucher

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H E S E

pour l'obtention du grade de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention : Électronique

présentée et soutenue par

Clément Foucher

Méthodologie de conception pour la virtualisation et le déploiement d'applications parallèles sur plateforme reconfigurable matériellement

Thèse dirigée par Alain Giulieri et co-encadrée par Fabrice Muller

soutenue le 24 octobre 2012

en présence du jury composé de :

M. Alexandre Nketsa	Professeur	Président du jury
M. Jean-Luc Dekeyser	Professeur des universités	Rapporteur
M. Guy Gogniat	Professeur	Rapporteur
M. Alain Giulieri	Professeur des universités	Directeur de thèse
M. Fabrice Muller	Maître de conférences	Co-encadrant

Résumé

Méthodologie de conception pour la virtualisation et le déploiement d'applications parallèles sur plateforme reconfigurable matériellement

Les applications auto-adaptatives, dont le comportement évolue en fonction de l'environnement, sont un élément clé des systèmes de demain. L'utilisation de matériel reconfigurable, combiné à la parallélisation des unités de calcul, permettent d'envisager de nouveaux niveaux de performances pour ces mêmes applications.

L'objectif de cette thèse est de mettre en place un ensemble d'outils permettant la description et le déploiement d'applications parallèles auto-adaptatives. Nous proposons à la fois un modèle d'application parallèle et une architecture de plateforme reconfigurable destinée au déploiement des applications conçues en utilisant ce modèle.

Notre modèle d'applications sépare le contrôle du calcul en isolant ce dernier en différents noyaux virtuels. Associée à une représentation du contrôle indépendante de la plateforme, la structure de l'application est donc totalement portable. Indépendamment de celle-ci, les noyaux de calcul peuvent être distribués selon plusieurs implémentations, selon la plateforme, mais également pour proposer différents niveaux de performances. Une couche de virtualisation permet de faire le lien entre la partie contrôle et les noyaux, en traduisant les ordres génériques en actions adaptées à l'implémentation.

Concernant la plateforme, nous proposons une architecture permettant l'intégration de ressources de calcul logicielles et matérielles, pouvant être implémentées tant statiquement qu'en utilisant du matériel reconfigurable. Cette architecture parallèle, inspirée du modèle des supercalculateurs, doit permettre d'utiliser tout type d'unités d'exécution et de matériel reconfigurable comme base matérielle pour la plateforme.

Design methodology for virtualization and deployment of parallel applications on reconfigurable hardware platform

Auto-adaptive applications, changing their behavior depending on environmental interactions, are a centerpiece of future computing systems. Moreover, the use of reconfigurable hardware in combination with processing units parallelization, allow us to consider reaching upper levels of performance for these applications.

The aim of this thesis is to develop a set of tools allowing description and deployment of parallel auto-adaptive applications. We propose both a model for parallel applications and a reconfigurable platform architecture to host deployment of applications described using this model.

Our application model separates control from computation by isolating the latter in virtual computation kernels. Together with a platform-independent application control representation, application structure is thus totally portable. Independently from the control, the computation kernels can be distributer using various implementations, depending on the platform, but also allowing proposing various performance factors. A virtualisation layer is used to link the control and the kernels, traducing generic orders into action depending on the implementation.

About the platform, we propose an architecture allowing integrating both software and hardware resources, either static or using reconfigurable hardware. This High Performance Computer-like parallel architecture should allow using any kind of processing unit and reconfigurable hardware as a hardware base for the platform.

Remerciements

Je tiens à remercier en premier lieu tous les membres du laboratoire LEAT qui m'ont chaleureusement accueilli et permis de réaliser mes recherches, toujours dans une bonne ambiance de travail. En particulier, l'équipe MCSOC, au sein de laquelle j'ai eu l'honneur de travailler, a toujours été conviviale et les relations cordiales.

Je remercie également Alain, mon directeur de thèse, ainsi que Fabrice, qui m'ont encadré tout au long de ces trois années de thèse, et m'ont initié aux rouages de la recherche scientifique. En particulier Fabrice, qui a été présent au quotidien, me guidant dans mes choix tout en me laissant maître de ceux-ci, m'a été d'une grande aide.

Enfin, je remercie ma famille et mes amis, qui m'ont soutenu dans les moments plus difficiles, m'apportant les instants de détente nécessaires lorsque la charge de travail était élevée.

Table des figures

Liste des tableaux

Table des matières

Glossaire

AES Advanced Encryption Standard.

AHB Advanced High-performance Bus.

AMBA Advanced Microcontroller Bus Architecture.

APB Advanced Peripheral Bus.

API Application Programming Interface.

ASIC Application-Specific Integrated Circuit – Circuit spécifique à une application.

AXI Advanced eXtensible Interface.

BRAM Bloc RAM.

CLB Configurable Logic Block – Élément programmable.

CPU Central Processing Unit – Unité de traitement principal / Processeur central.

CUDA Compute Unified Device Architecture.

DMA Direct Memory Access – Gestionnaire d'accès direct à la mémoire.

DSP Digital Signal Processor – Processeur de traitement de signal.

ELF Executable and Linkable Format.

FaRM Fast Reconfiguration Manager.

FIFO First In - First Out.

FPGA Field-Programmable Gate Array – Matrice d'éléments programmables.

FPU Floating Point Unit – Unité de calcul en virgule flottante.

GPGPU General Purpose GPU – GPU à vocation généraliste.

GPU Graphic Processing Unit – Jeu d'instruction.

HPC High Performance Computer – Superordinateur.

HPRC High Performance Reconfigurable Computer – Superordinateur reconfigurable.

HSDP Stream Dynamic Platform.

ICAP Internal Configuration Access Port.

IP Intellectual Property block – Bloc de propriété intellectuelle.

ISA Instruction Set Architecture – Jeu d'instruction.

LMM Lite Memory Manager.

LRU Least Recently Used.

LUT LookUp Table – Table de correspondance.

MIMD Multiple Instruction Multiple Data.

MoC Model of Computation – Modèle de calcul.

MPI Message Passing Interface.

MPICH The MPI CHameleon.

MPMC Multi-Port Memory Controller.

MPMD Multiple Program Multiple Data.

NNUS Nonuniform Node Uniform System.

NoC Network on Chip – Réseau sur puce.

NUMA Non-Uniform Memory Access – Accès mémoire non uniforme.

OpenCL Open Computing Language.

OpenMP Open Multi-Processing.

OS Operating System – Système d'exploitation.

PE Processing Element – Unité d'exécution.

PLB Processor Local Bus.

RDP Reconfiguration Dynamique Partielle.

SHP Software HPC Platform.

SIMD Single Instruction Multiple Data.

SMP Shared Memory Processing / Simultaneous MultiProcessing – Système à mémoire partagée.

SoC System on Chip – Système sur puce.

SoPC System on Programmable Chip.

SPMD Single Program Multiple Data.

SPoRE Simple Parallel platform for Reconfigurable Environment.

UART Universal Asynchronous Receiver Transmitter – Émetteur-récepteur asynchrone universel.

UMA Uniform Memory Access – Accès mémoire uniforme.

UNNS Uniform Node Nonuniform System.

XML Extensible Markup Language.

Chapitre 1

Introduction

Sommaire

Les systèmes de traitement des données numériques occupent une place de plus en plus importante en tant qu'éléments visibles de la vie quotidienne, mais également de manière sous-jacente en tant que composants des systèmes d'information régissant un nombre grandissant de domaines. Le traitement numérique des donnés utilise toute une variété d'approches pour arriver à ses fins, depuis le logiciel, consistant en l'utilisation de matériel générique permettant par programmation de répondre à différents besoins, jusqu'au matériel dédié à une opération, qui permet d'atteindre des performances bien supérieures de par sa conception spécifique. Le degré de parallélisme, correspondant au nombre de ressources de calcul, est également un élément déterminant de la puissance des systèmes.

L'intégration des ressources de calcul matérielles a longtemps été limitée par l'aspect statique de celles-ci. En effet, l'intégration d'un accélérateur matériel sous la forme d'un circuit électronique figé n'est utile que si l'on est sûr que celui-ci sera utilisé. Les circuits embarqués, destinés à une tâche particulière qui ne variera pas durant toute la vie du dispositif, sont de bons candidats pour intégrer ces composants. En revanche, il n'en est pas de même pour un système informatique générique, dont l'usage effectif dépend des besoins de son utilisateur. Néanmoins, l'arrivée plus récente du matériel reconfigurable, permettant d'implémenter divers accélérateurs matériels, a remis en cause cette position. Depuis, la souplesse introduite par cette technologie a permis de réaffirmer l'utilité du matériel aux cotés du logiciel, et la complémentarité de ces deux approches.

À l'origine, les systèmes informatiques grand public ne comportaient qu'une seule ressource centrale de traitement logiciel, éventuellement épaulée par divers accélérateurs matériels dédiés à des opérations particulières telles que le traitement du son ou de l'image. À l'opposé de cette approche, les supercalculateurs professionnels, destinés à des opérations extrêmement gourmandes en puissance de calcul, ont rapidement fait le choix de hauts degrés de parallélisme pour leur exécution. À l'heure actuelle, le choix du parallélisme semble s'imposer pour contenter les besoins croissants en puissance de calcul, face à la difficile tâche de l'augmentation de la puissance des ressources elles-mêmes. La reconfiguration matérielle, quant à elle, prend peu à peu sa place dans ces systèmes, mais n'a pas encore atteint le niveau de maturité suffisant pour le marché grand public, et reste cantonnée aux dispositifs particuliers, notamment dans le milieu de l'embarqué.

Outre les applications requérant de hautes performances, une autre catégorie est susceptible de tirer parti du matériel reconfigurable : les applications auto-adaptatives. Une application auto-adaptative, comme son nom l'indique, modifie son propre comportement en fonction de divers critères environnementaux. Pour un même traitement, l'application peut ainsi choisir parmi différents algorithmes selon la situation. Or, si en logiciel l'utilisation d'implémentations différentes est aisée, l'utilisation de matériel statique rend plus difficile de tels comportements : il faut en effet disposer de circuits pour toutes les implémentations possibles, sachant

qu'un seul sera actif à la fois. En termes de surface utilisée, une grande partie du composant est alors inactive à chaque instant, ce qui correspond à une perte de place. La reconfiguration matérielle résout ce problème en permettant d'utiliser une même surface pour les différentes implémentations, en reconfigurant celle-ci lors du changement d'algorithme.

Logiciel, matériel, degré de parallélisme, reconfiguration dynamique, auto-adaptativité... les critères de conception des systèmes sont nombreux et par voie de conséquence, l'hétérogénéité entre les systèmes existants est forte. Cette thèse s'inscrit dans le contexte de l'unification de ces différentes technologies, et œuvre à proposer un ensemble d'outils pour simplifier la gestion de systèmes hétérogènes. Différents niveaux sont à distinguer dans cette approche : la structure de l'application, la manière dont les implémentations interagissent entre elles et avec les éléments de contrôle, l'architecture de la plateforme d'exécution, etc. Ainsi, il faudra définir chacun des éléments de la vie d'une application, de sa conception à son exécution, afin de proposer des outils répondant à chacune des étapes de ce flot, pour tenter de proposer une solution pour la prise en compte de l'hétérogénéité dans les systèmes parallèles et supportant l'auto-adaptativité.

Dans cette introduction, je commence par développer un certain nombre de constats ayant porté mes motivations pour ce travail dans la section 1.1. Puis, je définirai les objectifs que je vise dans la section 1.2. Enfin, sur cette base, j'identifie les différents éléments à spécifier pour atteindre ces objectifs dans la section 1.3.

1.1 Constats et motivations

Le traitement automatisé de données a réellement pris son essor avec l'utilisation du logiciel, qui a permis pour une unique plateforme de calcul de supporter différents algorithmes. Cette flexibilité, séparant la nature matérielle de la fonctionnalité, a permis de développer des systèmes génériques et de les distribuer de plus en plus largement au fil de la miniaturisation.

Par la suite, l'introduction d'éléments de calcul matériels pour permettre une augmentation des performances sur le traitement spécifique de certains algorithmes a toujours suivi ce modèle centré sur le logiciel. Ainsi, à l'exception notable des périphériques embarqués, spécifiques à une fonctionnalité, les applications sont pour la plupart principalement logicielles. Ainsi, les calculs sont réalisés logiciellement jusqu'à rencontrer un noyau de calcul particulier dont l'exécution matérielle a été prévue. À ce point de l'exécution, le logiciel fait appel à l'accélérateur matériel, et lui soumet des données pour traitement. L'application reste donc pensée pour le logiciel, ne voyant le matériel que comme une unité d'appoint permettant d'accélérer certains calculs.

Tout ceci est question d'héritage : les couches se sont accumulées au cours de l'évolution des systèmes, et la remise en cause de certaines d'entre elles semble

une tâche inabordable tant les utilisateurs sont habitués à cette manière de faire. Ainsi, la simple démocratisation, récente, du parallélisme dans les systèmes par l'introduction de plusieurs ressources de calcul, est déjà un problème pour les développeurs : comment structurer nos applications de manière parallèle, alors que nous avons toujours fait cela séquentiellement. Ceci résulte en une sous-utilisation effective des capacités des systèmes actuels. Ainsi, alors que le parallélisme a été introduit dans les ordinateurs grand public depuis une dizaine d'années, le nombre d'applications tirant réellement parti de cette fonctionnalité reste relativement réduit, se confinant pour la plupart d'entre elles aux applications professionnelles, qui requièrent une grande puissance de calcul.

Le constat est le même concernant l'utilisation du matériel : les développeurs, qui ont toujours réfléchi à des applications logicielles, peinent à utiliser toutes les ressources matérielles disponibles. Un exemple flagrant concerne la vidéo. Alors que les accélérateurs matériels vidéos ont été parmi les premiers à être intégrés dans les ordinateurs grand public par le biais des cartes graphiques, certains logiciels largement utilisés ne supportent cette fonctionnalité que depuis peu. C'est ainsi que Flash [73] ou encore VLC [75] n'ont franchi le pas que depuis quelques années, quand ces applications, de par leur nature, auraient pu (ou du) être pensées nativement dans cette optique.

De la même manière, les développeurs ont mis longtemps à réellement tirer parti de la puissance des processeurs graphiques. En effet, ceux-ci ayant été conçus spécifiquement pour le calcul graphique, ils n'ont longtemps été utilisés que dans ce cadre limité. En réalité, leur puissance s'étend bien au-delà de ces calculs graphiques, capacité que l'on n'a commencé à exploiter que récemment.

Ainsi, les développeurs, habitués à utiliser toujours les mêmes outils, passent parfois à coté de ressources pourtant déjà présentes, alors même que la course à la puissance reste une motivation forte. Sur ce constat, on peut se demander ce qu'il en sera de la reconfiguration matérielle dans un futur proche. En effet, cette technique, longtemps limitée, commence à se développer réellement : aujourd'hui dans les systèmes embarqués, demain dans les ordinateurs grand public ? Le problème risque donc d'être une sous-utilisation des capacités de cette technique pendant une longue durée.

C'est donc fort de ces constats que je me suis fixé pour objectif de contribuer à la fois à la relation entre le logiciel et le matériel et à la prise en charge du parallélisme. Mon objectif est de rassembler tous ces éléments, incluant la reconfiguration matérielle et l'auto-adaptativité qui représentent des valeurs d'avenir, dans une nouvelle approche dont le but est de réellement tirer parti de la puissance disponible, en sortant du carcan de l'héritage sur les points précédemment cités.

1.2 Définition des objectifs

La problématique de la thèse est le déploiement d'un flot de conception pour des applications parallèles sur des architectures distribuées et reconfigurables dynamiquement. Mon objectif est de définir un ensemble d'outils destinés à créer et déployer de manière simple des applications parallèles tirant parti des capacités de reconfiguration dynamique du matériel. Dans une tendance globale à la diminution du temps de développement, mieux connu sous le nom de « time to market », la complexité du développement peut mener à certaines réserves quant à l'utilisation d'applications hétérogènes, voire à leur préférer des implémentations plus classiques bien que moins performantes. La simplification du travail d'implémentation sera donc l'un des objectifs principaux de mon travail, afin de renforcer l'attrait des applications et plateformes hétérogènes. Sur cette base, plusieurs axes de recherche se dessinent.

Tout d'abord, du point de vue de la conception de l'application. Lors du développement d'applications parallèles, la communication entre les unités hétérogènes représente souvent un point complexe : prise en considération de la nature de l'unité d'exécution, écriture éventuelle de pilotes pour communiquer avec le matériel... La prise en charge des échanges de données constitue donc une partie importante du travail de développement de ces applications. Sur la base de cette considération, nous identifions notre premier objectif, qui est de permettre le développement d'applications parallèles supportant différents types de ressources sans que le développeur ait besoin de gérer manuellement leurs interactions. Nous devrons donc définir une méthode de prise en charge automatique de celles-ci, de manière à alléger le travail du développeur, afin de renforcer l'attrait de ces applications.

Un autre aspect de la problématique concerne l'utilisation de la reconfiguration matérielle, et plus spécifiquement de la reconfiguration dynamique partielle. Cette dernière, en permettant la modification des circuits logiques contenus dans un design implémenté sur FPGA de manière dynamique, offre d'intéressantes possibilités d'adaptation. Néanmoins, la gestion de la reconfiguration partielle nécessite toute une gamme d'outils pour sa prise en charge, et alourdit considérablement le travail de développement. Si nous pouvions proposer une prise en charge transparente de celle-ci du point de vue du développeur de l'application, et donc éviter à celui-ci de devoir en connaître les mécanismes, son attrait se trouverait là aussi renforcé. Après tout, demande-t-on à un développeur logiciel travaillant sur un langage de haut niveau de savoir compiler un programme ?

Un autre axe d'intérêt concerne la réutilisation d'éléments développés antérieurement dans de nouvelles applications. Souvent au cœur des développements à l'heure actuelle, la réutilisation permet de s'affranchir de la conception de pans entiers du développement d'applications quand une fonctionnalité a déjà été développée par le passé. Or, si la réutilisation d'un élément matériel ne pose pas

de problème dans le cas d'une architecture similaire, l'utilisation dans une nouvelle forme d'architecture peut parfois ne pas être immédiate, demandant un effort d'adaptation. Ceci est particulièrement vrai pour la réutilisation de matériel statique dans une architecture dynamique, les composants n'étant pas prévus pour cette utilisation. Comme nous souhaitons proposer une solution directement utilisable, l'un des points essentiels de mon travail sera de proposer la réutilisation d'éléments statiques en leur ajoutant un caractère dynamique sans besoin d'adaptation de la part du développeur.

Par ailleurs, il faudra mettre en place une méthodologie destinée au développement d'applications tirant parti de tous ces outils. Ainsi, je devrai proposer une manière de construire une application utilisant le parallélisme, le matériel, y compris reconfigurable, et la réutilisation. Ceci permettra de proposer une méthode accessible pour le développement d'applications hétérogènes.

1.3 Identification des éléments à spécifier

Notre objectif étant de proposer un ensemble d'outils pour répondre à ces différents points, il nous faudra une solution verticale, allant de la conception de l'application parallèle hétérogène jusqu'à son exécution. Cet ensemble d'outils sera utilisé pour la mise en place d'un *flot* de conception et d'exécution. Celui-ci devra répondre à toutes les étapes de la vie d'une application, tout en permettant une compatibilité maximale avec la réutilisation d'éléments existants.

1.3.1 Modèle

Pour cela, nous proposerons en premier lieu une manière de représenter les applications parallèles combinant des éléments logiciels et matériels. En effet, la représentation de l'application est un élément déterminant dans sa conception, qui influe fortement sur la structure qui sera utilisée pour sa conception. Pour cela, il sera nécessaire de proposer un modèle adapté à la représentation de ce type d'applications. Ce modèle devra répondre à plusieurs points de notre problématique.

Tout d'abord, ce modèle devra être nativement parallèle. La prise en compte du parallélisme sera ainsi à la base de la représentation utilisée, et cette dernière devra permettre de représenter de hauts niveaux de parallélisme sans alourdir le modèle. Comme nous les verrons, plusieurs types de parallélisme existent, qu'il nous faudra supporter.

Par ailleurs, afin de supporter l'hétérogénéité, le modèle ne devra pas privilégier une nature par rapport à une autre. Ainsi, le modèle d'application devra supporter de manière indifférenciée les différentes possibilités d'implémentations, logicielles et matérielles, mais également statiques et reconfigurables.

1.3.2 Méthodologie de conception

Ce modèle devra être en accord avec l'élément suivant de notre flot : la méthodologie de conception. En définissant à la fois le modèle et la méthodologie, nous proposerons ainsi une équivalence directe entre les deux, permettant de passer de manière immédiate du modèle à l'implémentation qui en sera faite. Ainsi, nous définirons une méthodologie permettant de développer des applications parallèles hétérogènes, et d'intégrer celles-ci avec des éléments logiciels et matériels en proposant, pour chacun de ces types, le support statique et dynamique.

1.3.3 Plateforme d'exécution

Par ailleurs, la construction d'une application n'est rien s'il n'est pas possible de l'exécuter. Pour cela, nous mettrons en place les spécifications d'une plateforme d'exécution à même de supporter le déploiement des applications définies selon notre méthodologie. Toujours dans un souci de compatibilité avec les éléments existants, celle-ci s'inspirera de modèles existants, et permettra d'utiliser des éléments standards en tant que brique de base, tels des FPGAs et des processeurs.

Puis, afin de valider ce modèle, nous procèderons à deux implémentations de cette plateforme, chacune selon une orientation spécifique. Cela nous permettra de vérifier la pertinence du modèle, chacune de ces deux implémentations étant orientée vers la validation de points spécifiques. Ces deux implémentations seront chacune validée sur une application de test, qui nous permettront de confirmer le modèle sur des éléments concrets. Ces plateformes, outre les ressources de calcul, comprendront tout un environnement d'exécution pour la prise en charge de l'exécution des applications.

1.4 Structure du document

Ce document est structuré de la façon suivante : Dans le chapitre 2, nous présentons l'état de l'art en matière de parallélisme et de reconfiguration matérielle. Après avoir défini les termes importants, nous détaillerons les différents types de parallélisme existant, présenterons la technologie de reconfiguration matérielle et les relations entre les unités d'exécution.

Ensuite, le chapitre 3 nous permettra de spécifier les différents composants que nous avons détaillés dans la section précédente. Nous nous attellerons ainsi à définir un modèle d'applications parallèles hétérogènes, ainsi qu'une plateforme d'exécution.

Dans les chapitres 4 et 5, nous présenterons les implémentations issues de ces spécifications. Nous détaillerons ainsi les spécificités de chacune de ces implémentations, et nous procèderons à leur test afin de les valider.

Enfin, dans le chapitre 6, une synthèse du manuscrit sera proposée et nous nous intéresserons aux perspectives résultant de notre travail.

Chapitre 2

État de l'art

Sommaire

Nous avons identifié deux axes majeurs dans la problématique : l'utilisation de ressources matérielles, notamment reconfigurables, et le parallélisme. Ces deux éléments sont des facteurs permettant d'augmenter la puissance de calcul. Ils sont étroitement liés puisque les éléments matériels sont intrinsèquement parallèles.

Dans un premier temps, il sera nécessaire de définir les termes essentiels concernant les systèmes de calcul parallèles, tant matériels que logiciels. Il faudra notamment préciser ce que l'on entend par *matériel* et *logiciel*, leur définition pouvant varier selon les sources. Sur cette base, nous étudierons les architectures des systèmes de calcul parallèles et leurs ressources, ainsi que les environnements permettant de développer des applications.

2.1 Définitions

Afin de définir les différents termes nécessaires à la compréhension du domaine abordé, nous nous basons sur un exemple de système de traitement des données. Nous prendrons pour cela l'exemple d'un ordinateur personnel actuel.

Un ordinateur est constitué d'un ensemble de composants électroniques, notamment des mémoires et des unités d'exécution. La principale ressource de calcul est le processeur central (Central Processing Unit – CPU), rassemblant plusieurs cœurs qui sont autant d'unités d'exécution (Processing Element – PE). Ces PEs peuvent réaliser différents calculs par le biais de programmes, qui sont des ensembles d'instructions exécutées séquentiellement. Ces programmes sont codés selon un jeu d'instruction (Instruction Set Architecture – ISA) spécifique à une famille de processeurs.

2.1.1 Logiciel et matériel

Cette observation nous fournit une première définition du matériel et du logiciel : une unité d'exécution est un élément physique, c'est-à-dire du matériel ; un programme est une donnée traitée par un élément matériel, et constitué d'instructions logicielles.

Mais dans un ordinateur, le CPU n'est pas la seule unité d'exécution. On trouve généralement un ensemble de composants dédiés à des traitements particuliers, tels que le son ou l'image. Par exemple, il est fréquent d'inclure des composants matériels dédiés au décodage de formats vidéos répandus. Ainsi, lors de la lecture d'une vidéo dont le format est pris en charge par le matériel, son décodage ne sera pas confié au CPU, mais à l'unité d'exécution concernée.

L'utilisation d'un composant dédié permet de meilleures performances que l'utilisation d'un CPU, car il est construit dans le but de décoder ce type de flux, et bénéficie donc d'une architecture logique spécifiquement conçue pour cette opération, là où un CPU doit être capable de réaliser tout type d'opération. Une unité

d'exécution dédiée n'est pas programmable, contrairement à un CPU, du fait de son but précis. Il ne s'agit donc pas d'une unité d'exécution logicielle, et on la qualifiera alors de PE matériel.

Ainsi, l'acceptation générale des termes logiciel et matériel dans les systèmes de traitement des données se rapporte généralement à la nature d'un PE. Une exécution logicielle nécessitera un programme qui sera exécuté sur un PE de type CPU, quand une exécution matérielle se passera d'instructions, car le PE est déjà conçu dans un but précis.

De plus, le CPU n'est généralement plus le seul PE logiciel disponible dans un ordinateur. Les processeurs graphiques (Graphic Processing Unit – GPU) sont des PEs logiciels possédant une structure particulière très différentes des CPUs. Là ou un CPU peut posséder plusieurs cœurs puissants en nombre réduit, un GPU possède un très grand nombre de cœurs ayant une structure simplifiée. En effet, destinée au calcul graphique, dans lesquels les traitements portent sur un grand nombre de pixels, cette structure permet une parallélisation des calculs afin de traiter simultanément plusieurs pixels.

Mais cette structure parallèle des GPUs est également adaptée à d'autres types de calculs parallèles que les traitements graphiques. Pour cette raison, on utilise de plus en plus les GPUs en tant que GPUs à vocation généraliste (General Purpose GPU – GPGPU), afin d'accélérer les calculs par l'exploitation de leur parallélisme intrinsèque.

2.1.2 Application et plateforme

Mais un ensemble de PEs, logiciels et/ou matériels, n'est pas suffisant pour traiter des données. Un certain nombre d'autres éléments logiques sont nécessaires : mémoires pour stocker les données et les instructions, bus permettant l'interconnexion de ces éléments, etc. De plus, un système complexe est généralement piloté par une couche logicielle, composée principalement d'un système d'exploitation (Operating System – OS).

L'ensemble de ces ressources matérielles et de l'éventuelle couche logicielle permettant son utilisation constitue une *plateforme*. Chaque plateforme a des caractéristiques particulières : ISA des PEs logiciels, nature des PEs matériels disponibles pour réaliser des opérations particulières, type de l'OS, etc.

Une plateforme est destinée à l'exécution d'*applications*. En langage courant, une application désigne généralement un programme informatique employé par un utilisateur pour mener à bien certaines tâches. Il existe donc de nombreuses applications employées quotidiennement sur un poste de travail : applications bureautiques, navigateurs internet, environnements de programmation, etc.

Dans un système embarqué, l'application désigne en général le processus de traitement complet mis en œuvre par le système. Contrairement à une application

au sens informatique, celle-ci peut donc ne pas avoir, ou très peu, d'interaction directe avec l'utilisateur, mais plutôt avec l'environnement. Par exemple, pour une application embarquée dans une voiture récente, différents processus sont enclenchés automatiquement sans interaction avec un utilisateur : mesure de vitesse, du niveau de l'essence, affichage des informations sur l'écran du tableau de bord, etc. Ces opérations sont réalisées par le biais de capteurs intégrés à la plateforme.

Dans le domaine de l'embarqué, l'application est ainsi souvent indissociable de la plateforme car elle dépend de composants matériels spécifiques. On utilise le terme de *circuit spécifique à une application* (Application-Specific Integrated Circuit – ASIC) pour définir de telles plateformes.

Dans ce cas, l'application n'est donc pas simplement logicielle, mais intègre également des éléments matériels de la plateforme. Il est même possible de rencontrer des ASICs ne disposant pas du tout de composante logicielle, l'application étant uniquement composée de l'ensemble matériel.

En règle générale, une application consiste donc en un traitement de données, c'est-à-dire une utilisation de données entrantes pour produire des données sortantes. Ces données peuvent être générées par l'utilisateur, par exemple en tapant sur un clavier, provenir de capteurs, ou encore être issues d'un appareil de stockage. Selon les cas, elle peut être complètement logicielle, complètement matérielle, ou le plus souvent composée d'un ensemble des deux.

Nous nous intéressons particulièrement aux applications parallèles, c'est-à-dire aux applications destinées à fonctionner sur des plateformes disposant de plusieurs voire de nombreux PEs. On utilisera le terme processus pour désigner une partie d'une application qui s'exécute sur un PE. Une application purement séquentielle sera donc composée d'un seul processus à la fois, là où une application parallèle en utilisera plusieurs simultanément.

2.2 Parallélisme

Le terme parallélisme fait référence à un traitement simultané de plusieurs calculs. On rencontre deux niveaux principaux de parallélisme : le parallélisme interne aux PEs et la mise en parallèle de plusieurs PEs. Le premier permet d'accélérer le traitement d'un processus, quand le second permet de traiter plus de processus simultanément.

Les PEs matériels sont intrinsèquement parallèles. En effet, dédiés de par leur conception à un traitement particulier, l'algorithme à réaliser est connu. Il est alors possible d'identifier les calculs de l'algorithme pouvant s'exécuter en parallèle, et de les implémenter ainsi de manière à effectuer plusieurs opérations à chaque cycle d'horloge.

Dans le cas logiciel, le parallélisme interne s'exprime d'une manière différente : le *pipeline*. Le principe du pipeline consiste à séparer le traitement d'une instruction

en plusieurs étapes exécutées chacune à un cycle d'horloge. En effet, un processus logiciel déployé sur un PE est constitué d'un ensemble d'instructions à exécuter séquentiellement. Ainsi, lorsque la première instruction a terminé la première étape, il est possible d'injecter l'instruction suivante à cette étape. À chaque cycle, les différentes étapes sont donc exécutées simultanément, chacune sur une instruction différente.

Notons qu'il est également possible d'exécuter plusieurs processus de manière pseudo-simultanée sur un seul PE, en allouant des tranches de temps à chacun. Ce découpage temporel (« time slicing »), réalisé relativement rapidement à l'échelle humaine permet de simuler une exécution simultanée de différentes applications. Si le time slicing est toujours très utilisé en logiciel, le parallélisme réel, utilisant plusieurs PEs, est depuis longtemps utilisé dans le domaine scientifique, et se développe de plus en plus dans les ordinateurs personnels grâce aux architectures multicœurs et à l'utilisation des GPGPUs pour réaliser des opérations généralistes. C'est ce dernier type de parallélisme qui nous intéresse ici, c'est-à-dire l'exécution simultanée de plusieurs processus au moyen de la mise en parallèle de plusieurs PEs.

Dans cette section, nous distinguons deux principaux niveaux de parallélisme utilisés dans les systèmes de traitement des données, en s'intéressant pour l'instant uniquement aux systèmes à PEs logiciels, les plus répandus dans les plateformes de calcul parallèle généralistes. Le passage aux PEs matériels se fera dans la section 2.3. Ces deux niveaux de parallélisme concernent les systèmes à mémoire partagée, étudiés en 2.2.1 et ceux à mémoire distribuée, analysés en 2.2.2. Ces deux types d'architectures sont décrits par Hager et Wellein [39] dans une étude très structurée, partant d'un processeur simple, et ajoutant peu à peu les différentes techniques de parallélisme pour aboutir aux systèmes complexes à très haut degré de parallélisme.

2.2.1 Parallélisme local : mémoire partagée

Le premier niveau de parallélisme concerne les systèmes à mémoire partagée (Shared Memory Processing ou Simultaneous MultiProcessing – SMP). Il s'agit de systèmes fortement couplés, dans lesquels la mémoire principale est utilisée conjointement par les différents PEs [14]. Un seul système d'exploitation gère tous les PEs et répartit la charge de travail entre eux. La mémoire étant commune, une méthode de communication possible entre les différents PEs consiste à utiliser celle-ci pour stocker les données à transmettre. La figure 2.1 illustre un exemple de système SMP.

On trouve différents niveaux de couplage entre les PEs de ces systèmes. Les PEs peuvent être situés sur la même puce, ou bien physiquement isolés. Selon le cas, on parle alors de puces multicœurs ou de systèmes multiprocesseurs. Bien entendu, un système multiprocesseur peut être à base de puces multicœurs. La

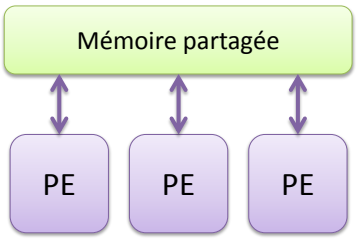


Figure 2.1 – *Représentation d'un système SMP.*

conception intégrée d'une puce multicœur apporte un avantage par rapport à la mise en parallèle de plusieurs puces, en permettant notamment un partage de certains niveaux de cache.

Le parallélisme au sein d'une puce est généralement homogène, ce qui signifie que tous les PEs sont similaires. C'est ainsi les cas des processeurs grand public à architecture x86 telle la famille des *Cores* de Intel et *Athlon* ou *Phenom* de AMD. Il est néanmoins possible de réaliser du multicœur hétérogène. C'est le cas notamment du Cell Broadband Engine (Cell B.E.) [43]. Le processeur Cell B.E. est constitué d'un cœur principal, le Power Processor Element (PPE) associé à 8 PEs secondaires, les Synergistic Processor Elements (SPEs). Il existe donc une hiérarchie entre le PPE et les SPEs. Le PPE est destiné à exécuter l'application elle-même, et délègue des traitements de données aux SPEs.

2.2.1.a Cohérence de cache

L'un des problèmes inhérents aux systèmes SMP est la cohérence des données. En effet, les processeurs utilisent abondamment le cache afin de limiter les échanges avec la mémoire centrale, sources de délais. Or, l'utilisation du cache a pour effet l'obsolescence des données stockées en mémoire centrale tant que les données en cache n'ont pas été synchronisées. Il est donc nécessaire de prévoir un mécanisme de cohérence de cache entre les différents PEs d'un système SMP pour éviter les problèmes.

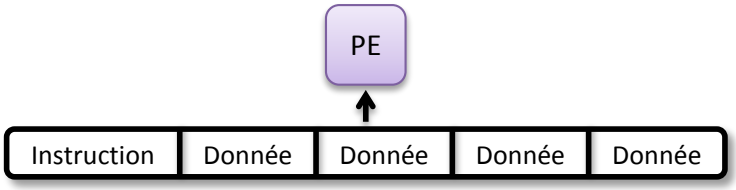
L'intégration de plusieurs unités sur la même puce permet un partage de certains niveaux de cache, ce qui simplifie la cohérence des données. Il est également possible de rendre des caches cohérents via l'utilisation de bus dédiés. Par exemple, AMD utilise un bus HyperTransport pour assurer la cohérence de cache des processeurs Opteron [49]. Les systèmes SMP sont à accès mémoire uniforme (Uniform Memory Access – UMA) [39], car l'accès à une donnée stockée en mémoire centrale est réalisé à temps constant quel que soit l'emplacement de la donnée dans la mémoire.

Il est également possible de rassembler plusieurs systèmes SMP, et donc plusieurs mémoires centrales, et de définir une mémoire virtuelle rassemblant ces différentes mémoires. L'accès à un emplacement de cette mémoire virtuelle dépend alors de l'emplacement physique, et peut prendre un temps d'accès différent selon que

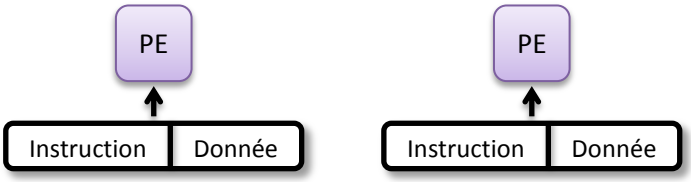
celle-ci est locale ou non. On parle alors de systèmes à accès mémoire non uniforme (Non-Uniform Memory Access – NUMA). Il est possible de rendre ces systèmes cohérents en termes de cache, là encore en utilisant des bus dédiés. On parle alors de système NUMA à cohérence de cache (Cache-Coherent NUMA – ccNUMA) [39].

2.2.1.b Type de parallélisme

On distingue deux types de parallélisme logiciel : parallélisme de données ou d'instruction [78]. Ceux-ci sont présentés sur la figure 2.2. Le parallélisme de don-



(a) *Une instruction est envoyée avec plusieurs données sur une unité SIMD*



(b) *Chaque unité d'un système MIMD reçoit des instructions différentes*

Figure 2.2 – *Les deux types de parallélisme logiciel.*

née (Single Instruction Multiple Data – SIMD) consiste à traiter plusieurs données en une seule instruction. Ce type de parallélisme est en général implémenté au sein même d'un PE. C'est par exemple le cas des SPEs du processeur Cell B.E., qui sont des unités SIMD de 128 bits. Une telle unité SIMD pourra ainsi proposer des instructions pour traiter simultanément deux mots de 64 bits, quatre mots de 32 bits, et ainsi de suite. Il est également possible de distribuer une simple instruction à plusieurs PEs simultanément afin de réaliser du SIMD. Le SIMD est généralisable au SPMD (Single Program Multiple Data) quand plusieurs unités exécutent simultanément un même programme. Néanmoins, contrairement au SIMD, où les mêmes instructions sont exécutées au même moment, deux PEs fonctionnant en SPMD peuvent en être à un état d'avancement différent dans l'exécution du programme à un instant donné.

Le parallélisme d'instruction (Multiple Instruction Multiple Data – MIMD) consiste à exécuter simultanément des instructions différentes sur des PEs différents. Généralisé au MPMD (Multiple Program Multiple Data), le parallélisme d'instruction consiste en un parallélisme indépendant entre plusieurs PEs.

Pour résumer, le parallélisme de données consiste à traiter des données différentes, mais sur un même algorithme, tandis que le parallélisme d'instruction (ou d'application à un plus haut niveau) met en œuvre des algorithmes différents.

2.2.1.c Types d'unités d'exécutions

Si les CPUs sont généralement au centre des traitements de données logiciels, il existe d'autres PEs logiciels. Ainsi, par exemple, les premiers superordinateurs utilisaient des PEs vectoriels, traitant principalement des instructions de type SIMD.

Par ailleurs, outre les CPUs multicœurs, la recherche s'oriente de plus en plus vers les unités d'exécutions à nombreux cœurs (« manycore ») [41, 88, 82]. Les unités manycore utilisent un paradigme différent du SMP, celui-ci montrant ses limites au-delà d'une dizaine de cœurs [3]. En effet, l'engorgement de l'accès à la mémoire, commune aux différents PEs en SMP, devient rapidement un facteur limitant. Pour permettre la continuité de l'augmentation du nombre d'unités d'exécutions, les manycore se basent en général sur un paradigme équivalent à celui des réseaux. Ainsi, chaque PE dispose d'une mémoire locale, et est connecté aux autres par un bus orienté réseau, que l'on nomme alors *réseau sur puce* (Network on Chip – NoC) [6]. Plusieurs puces manycore ont été développées, principalement à des fins de recherche. On citera notamment le Single-Chip Cloud Computer (SCC) [44], une puce développée par Intel. Celle-ci est composée de 24 éléments comportant chacun deux PEs, reliés par un NoC. Chaque élément a une structure équivalent à un processeur bicœur, incluant sa propre mémoire cache.

Un autre composant est de plus en plus utilisé en tant que PE logiciel : les processeurs graphiques. À la base destinés au calcul graphique, comme leur nom l'indique, il s'est par la suite avéré que ces processeurs pouvaient avantageusement être utilisés pour d'autres types de calculs. Utilisés en tant que GPGPUs, les processeurs graphiques ouvrent de nouvelles perspectives de puissance en proposant une architecture consistant en une matrice de très nombreuses micro-unités d'exécutions [13]. Ce type de structure permet de réaliser avantageusement des opérations sur des vecteurs. En dépit de la présence de nombreux micro-PEs en leur sein, on traitera tout de même ces unités comme un PE unique, celles-ci ne pouvant exécuter plusieurs processus indépendamment.

2.2.2 Mise en parallèle de systèmes à mémoire partagée

Ainsi, un premier niveau de parallélisation des PEs utilise une mémoire commune et gérée par un système d'exploitation unique, même si, comme indiqué, les puces manycores commencent à se développer, qui adoptent un paradigme différent. Afin d'accéder à un plus haut niveau de parallélisme, une solution consiste à mettre en parallèle plusieurs systèmes SMP, augmentant naturellement le parallélisme, comme présenté sur la figure 2.3.

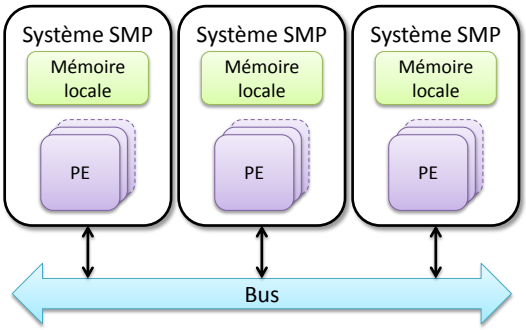


Figure 2.3 – *Représentation d'un système à mémoire distribuée.*

Les systèmes de ce type sont constitués de *nœuds*, chacun étant un système SMP. Les nœuds sont reliés entre eux par des bus, en général de type réseau [80]. La communication entre nœuds ne pouvant plus passer par une mémoire commune, inexistante sauf à mettre en place une topologie NUMA, on utilise le plus souvent une communication par échange de messages entre les nœuds. C'est cette même structure que l'on retrouve dans les manycores, mise à l'échelle d'une puce.

Cette structure en nœuds distribués sur un réseau est en général adoptée pour la plupart des *superordinateurs* (High Performance Computer – HPC). Une définition de la catégorie HPC est proposée par Oyanagi [61] : "Supercomputers are regarded as the computers which have an order of magnitude higher performance." (Les superordinateurs sont des ordinateurs dont les performances se situent un ordre de grandeur au dessus [de celles des ordinateurs classiques]).

Un historique retraçant l'évolution des HPCs depuis les premiers, à base de processeurs vectoriels, a été réalisé par Strohmaier [80] en 1999. Depuis cette date, les principales évolutions dans les HPCs concernent l'introduction de GPGPUs [38] ainsi que de matériel reconfigurable [36]. Un classement des HPCs les plus performants est tenu et actualisé tous les six mois : le top 500 [69]. Dongarra, créateur du test Linpack [21], utilisé dans sa variante High Performance Linpack (HPL) pour classer les ordinateurs au sein du top 500, présente régulièrement un rapport technique retraçant les évolutions de ce classement [15, 16, 17, 18, 19, 20]. En étudiant ces rapports, Iushchenko a formulé l'observation suivante [46] : Un ordinateur classé au sommet du top 500 en terme de performances se retrouve dernier après 6 à 8 ans. La puissance d'un ordinateur dernier au classement devient comparable à celle d'un ordinateur personnel après 8 à 10 ans. On note donc un délai de 14 à 18 ans avant qu'un ordinateur personnel atteigne la puissance du superordinateur classé au sommet du top 500.

2.2.3 Langages parallèles

Afin d'exploiter ces architectures parallèles, il est nécessaire de disposer des outils adéquats. Cela passe notamment par un langage de programmation disposant

des structures nécessaires pour exprimer le parallélisme. Le choix du langage est déterminant dans la phase de construction de l'application. En effet, la syntaxe même du langage influence la méthode de construction de l'application, et ce particulièrement dans le cas d'applications parallèles. Skillicorn [78] dira même que chaque langage de programmation est un modèle, car chacun propose une vue simplifiée du matériel sous-jacent. Cette proposition met bien en avant la prépondérance du choix du langage, fournissant un modèle de calcul (Model of Computation – MoC) particulier, par rapport aux besoins de l'application et au matériel sur lequel elle sera exécutée. Ce choix est d'autant plus déterminant que, dans certains cas, il peut restreindre la portabilité de l'application, c'est-à-dire sa capacité à être exécutée sur différentes plateformes.

À l'origine, la plupart des langages de programmation ont été conçus pour des systèmes à PE unique, et donc pour représenter des applications séquentielles. Le développement du parallélisme, dans les HPCs puis plus récemment dans les ordinateurs personnels a encouragé la création de langages parallèles. Deux méthodes ont été employées pour permettre cette évolution : faire évoluer des langages séquentiels existants, ou créer de nouveaux langages. Chaque méthode a ses avantages et ses inconvénients.

Ainsi, partir de langages existants permet de préserver les compétences des développeurs. En effet, maîtriser parfaitement un langage, et notamment toutes les petites astuces d'optimisation, nécessite une grande pratique. Développer une nouvelle fonctionnalité (ici, le parallélisme) sans toucher aux fondamentaux permet de réduire la période d'apprentissage et d'obtenir très rapidement de bons résultats, là où l'utilisation d'un nouveau langage va nécessiter une période de prise en main. Cette période peut être relativement longue avant de développer une vitesse d'écriture et une propreté de code équivalentes à celles que l'on possédait dans l'ancien langage.

À l'inverse, repartir de zéro en créant un langage *optimisé* pour une fonctionnalité permet de s'affranchir de toute contrainte, et donc potentiellement d'arriver à un meilleur résultat que par l'adaptation d'un langage existant. En effet, la syntaxe d'un langage séquentiel n'est pas forcément facilement applicable à une généralisation parallèle. Le résultat peut donc être des performances inférieures à un langage dédié, ou bien une plus grande complexité de développement pour arriver au même résultat.

La limite entre ces deux approches est parfois difficile à définir, notamment pour des langages présentés comme « nouveaux », et qui reprennent une syntaxe existante.

2.2.3.a Extensions parallèles de langages séquentiels

Les langages les plus répandus étant le C [50] (et son extension objet C++ [81]) et le Fortran [55, 62], ceux-ci disposent de plusieurs extensions destinées à implé-

menter le parallélisme. Certaines extensions ciblent même les deux langages simultanément, proposant en ensemble d'extensions disponibles à la fois en C et en Fortran.

Extensions parallèles au langage C/C++

Nous traiterons indifféremment le C et le C++ car ces deux langages sont intrinsèquement liés et partagent une syntaxe commune. Plus que de simples extensions, les deux langages que nous allons présenter, OpenCL et CUDA, sont de véritables environnements. En effet, en plus d'un ensemble de bibliothèques et de structures de code ajoutés au langage, ils nécessitent d'être explicitement supportés par la plateforme d'exécution. Ces deux environnements de développement permettent d'exploiter plusieurs types de PEs logiciels, et notamment des GPGPUs.

Le premier, *Open Computing Language* (OpenCL) [72, 83] est composé de deux éléments : une bibliothèque permettant la gestion de la plateforme et un langage permettant d'implémenter les algorithmes traitant les données. L'application est donc composée de deux couches, comme présenté sur la figure 2.4.

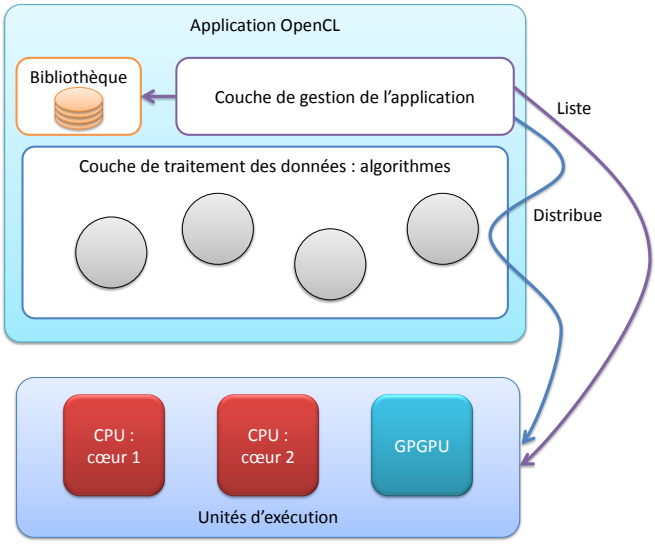


Figure 2.4 – *Couches d'une application OpenCL.*

La couche la plus basse consiste donc en l'implémentation des algorithmes de traitements. Celle-ci est écrite en OpenCL-C, un langage dérivé du C et adapté à l'expression du parallélisme. Cette couche permet d'écrire du code destiné à s'exécuter sur différents types d'unités d'exécution, et particulièrement les GPGPUs. Il est également possible de cibler des PEs plus spécifiques, tel le Cell B.E. [28] ou des processeurs de traitement de signal (Digital Signal Processor – DSP).

La seconde couche, de plus haut niveau, s'exécute spécifiquement sur le processeur hôte de la plateforme. Cette couche, écrite en C, permet la gestion des différents

codes destinés à s'exécuter sur les unités d'exécutions via une bibliothèque (Application Programming Interface – API). Celle-ci permet de lister les ressources disponibles sur la plateforme, et ainsi d'utiliser les implémentations adéquates.

OpenCL est de plus en plus utilisé pour autoriser l'exploitation des unités d'exécutions GPGPUs, notamment dans les ordinateurs personnels [22].

Un concurrent à OpenCL est *Compute Unified Device Architecture* (CUDA) [56]. CUDA permet, via l'utilisation de *C for CUDA*, de programmer des cœurs CUDA (CUDA cores) en donnant accès à leur jeu d'instruction. Les CUDA cores ne se trouvant actuellement que dans les GPGPUs de marque Nvidia, cela restreint considérablement les plateformes compatibles, à la fois sur le type et sur la marque. Néanmoins, cette spécialisation permet une forte optimisation, ce qui en fait un bon choix lorsque la portabilité n'est pas nécessaire. Par exemple, en novembre 2010, est entré en tête du classement top 500 la plateforme Tianhe-1A, disposant de 7 168 GPGPUs Nvidia, ce qui en fait une machine de choix pour une application CUDA.

Extensions parallèles au langage Fortran

Langage de prédilection pour exprimer des opérations mathématiques, le Fortran a vu plusieurs extensions développées pour permettre une expression du parallélisme.

L'extension High Performance Fortran (HPF) [64] a été développée avec pour objectif d'ajouter une manière efficace d'exprimer la concurrence d'une application sans sacrifier la portabilité. L'extension ajoute un ensemble de directives compilateur spécifiques (« pragmas ») sous la forme de commentaires spéciaux pour donner des instructions sur le parallélisme. Le fait de présenter ces instructions en tant que commentaires permet de conserver une compatibilité avec les compilateurs ne supportant pas le HPF, qui ignoreront tout simplement ces directives. Les pragmas HPF se concentrent principalement sur les tableaux en contrôlant leur alignement sur les différentes unités de calcul. Par exemple, pour une opération élément à élément sur deux tableaux, il faudra préciser que l'élément n du premier tableau doit se situer sur la même unité de calcul que l'élément m du second tableau pour permettre l'opération. Le fait de préciser l'alignement respectif de plusieurs tableaux permet ensuite une gestion automatique de la distribution des données à l'exécution. À cet ensemble de pragmas s'ajoutent des directives spécifiques facilitant les opérations sur les tableaux comme *FORALL*, qui permet d'agir sur chaque élément d'un tableau. On notera que l'utilisation de ces constructions, contrairement aux pragmas, introduit une rupture de compatibilité avec le standard Fortran. Néanmoins, en raison du développement de l'intérêt pour le parallélisme, une partie de l'extension HPF a par la suite été introduite au standard Fortran 1995, résolvant certains problèmes de compatibilité.

Une autre extension reconnue au Fortran fut le Co-Array Fortran (CAF) [57]. Le CAF consiste en une très simple modification de la syntaxe Fortran : l'ajout d'une dimension supplémentaire dans les tableaux. Les tableaux, qui en Fortran standard se définissent sous la forme *Tab(n)* deviennent *Tab(m)[p]*. La dimension indiquée entre crochet fait explicitement référence au numéro du PE. Ainsi, la syntaxe ci-dessus fait référence à l'élément m située sur le PE p. Cette extension permet une expression très simple du parallélisme, et fut elle aussi intégrée au standard Fortran 2008 par la suite.

On remarque donc que ces deux extensions parallèles au langage Fortran ont été intégrées au standard, démontrant l'intérêt de la communauté pour le calcul parallèle.

Extensions parallèles communes à C et à Fortran

Mais pourquoi choisir entre ces deux langages très largement exploités, quand il est possible de développer une extension pour les deux ? C'est ainsi que deux des principales extensions pour le parallélisme ont été développées à la fois pour le C et pour le Fortran. Ces deux extensions sont MPI et OpenMP, qui apportent deux visions différentes et complémentaires de l'architecture des plateformes.

OpenMP, pour *Open Multi-Processing* [12, 42], est un ensemble de pragmas spécifiant la structure parallèle de l'application. Une illustration du principe d'OpenMP est présentée en figure 2.5. Le développement en OpenMP consiste dans un premier

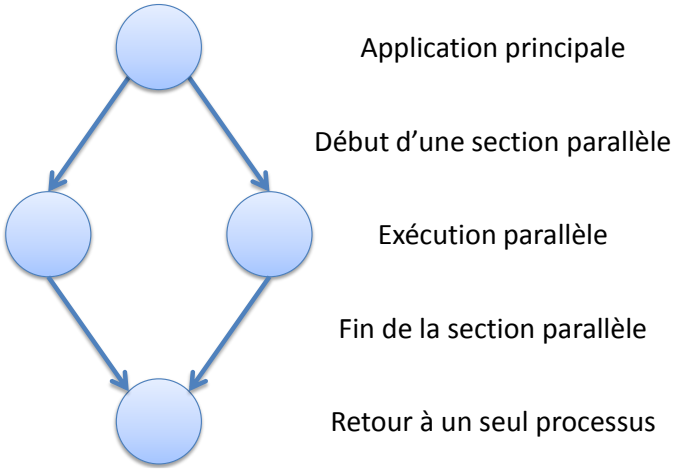


Figure 2.5 – *Structure d'une application OpenMP.*

temps à construire une application de la même manière qu'une application séquentielle. Ensuite, l'ajout des directives permet d'indiquer au compilateur quels éléments de l'application peuvent être exécutés en parallèle. Par exemple, une boucle peut être complétée d'instructions demandant une exécution simultanée de plusieurs itérations différentes. Ainsi, par l'ajout de ces instructions pour le compilateur, le programmeur construit le parallélisme de son application.

Par la suite, le compilateur met en place ce parallélisme sur un modèle *fork/-join* (littéralement, séparation/jonction) avec communication par mémoire partagée. Les applications OpenMP sont donc destinées aux systèmes SMP, et sont constituées d'un nombre variable de processus. En effet, le modèle fork/join consiste en un processus unique qui se « sépare » en plusieurs lors du début d'une section parallèle, pour se « rejoindre » à la fin. OpenMP permet donc une gestion de la communication implicite, c'est-à-dire invisible pour le programmeur, et gérée automatiquement par le compilateur. On appelle ce modèle la *vue globale*, car l'application est vue dans son ensemble plutôt que chaque processus indépendamment. Il reste possible d'utiliser OpenMP sur un système à mémoire distribuée en utilisant le paradigme NUMA vu précédemment.

Une approche opposée à OpenMP est proposée par Message Passing Interface (MPI) [40]. MPI se présente sous la forme d'une bibliothèque permettant la communication entre processus, dont le principe est présenté en figure 2.6. La bibliothèque

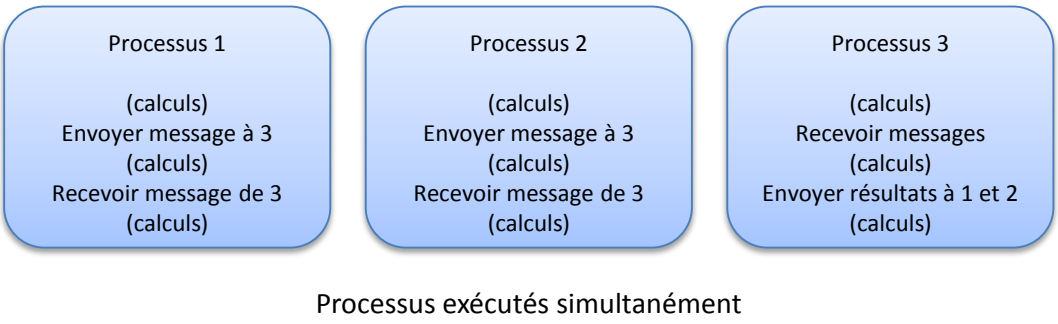


Figure 2.6 – *Structure d'une application MPI.*

propose des appels à des routines permettant à un processus d'envoyer et recevoir des messages à ses pairs. Il s'agit ici d'une autre vision d'une application, que l'on qualifiera de *vue locale*, car l'on détaille le fonctionnement de l'application du point de vue d'un processus et de sa communication avec les autres. L'approche par échange de message ne repose pas sur de la mémoire partagée mais est plutôt adaptée à un système composé de nœuds reliés par un réseau, bien qu'il reste possible de faire transiter les messages par la mémoire partagée.

MPI n'est pas à proprement parler une extension d'un langage, mais consiste en la définition d'un ensemble de directives de communications. Ces directives de communications sont de plusieurs natures : communication entre deux ou plusieurs processus, envoi ou réception de données. On trouvera particulièrement des fonctions de *réduction*, permettant de rassembler des données provenant de plusieurs nœuds sur un seul, par exemple par accumulation.

Les implémentations de MPI, sous la forme de bibliothèques, constituent les véritables extensions des langages, en donnant forme à la spécification. De nombreuses implémentations sont disponibles [37, 34]. Pour les superordinateurs, chaque con-

structeur propose en général sa propre implémentation de MPI, plus particulièrement adaptée à sa plateforme, et assure ainsi des performances proches de l'optimal pour les échanges MPI. Mais ces implémentations, propriétaires, ne sont pas librement disponibles et utilisables, et de toute façon ne seraient pas forcément utilisables sur d'autres plateformes.

Il existe également des implémentations libres et gratuites de la spécification MPI. Leur code source étant fourni, cela permet de les adapter à ses besoins si nécessaire, par exemple à une nouvelle plateforme qui ne serait pas compatible, ou sur laquelle les performances seraient mauvaises. L'implémentation libre de MPI la plus utilisée est MPICH (The MPI CHameleon – Le caméléon MPI, soulignant la portabilité de l'implémentation) [37]. MPICH est à l'opposé des implémentations constructeurs, car celle-ci n'est pas liée au un matériel, et assure une utilisation sur différentes plateformes.

2.2.3.b Langages nativement parallèles

Les langages nativement parallèles ont été développés avec comme contrainte le support du parallélisme.

L'un des principaux langages parallèles est Ada, conçu pour répondre à un cahier des charges de 1978 du département de la défense américain (Department of Defense – DoD), le *Steelman language requirements* [85]. Le Steelman, mis en place afin de choisir un langage unique de remplacement aux très nombreux langages utilisés par le DoD, axait ses spécifications sur la fiabilité, notamment dans un environnement parallèle. Ainsi, conçu particulièrement pour les systèmes embarqués, le MoC d'Ada représente une application sous la forme de modules concurrents, ce qui le rend parallèle par nature.

Le *Z-level Programming Language*, ou ZPL [11], ne se base pas sur un langage existant, mais permet de générer du code C. ZPL possède une syntaxe indépendante, opérant sur des *régions* de tableaux. Cette syntaxe permet de représenter les communications de manière concise, là où une syntaxe en C/MPI doit explicitement présenter ces échanges. Ce code est ensuite transformé par un compilateur en code C utilisant MPI et des bibliothèques propres, pour permettre une compilation classique avec un compilateur C. L'utilisation du ZPL permet donc d'obtenir les mêmes performances qu'un code C tout en utilisant une représentation concise.

2.2.3.c Combinaisons de langages parallèles

Comme nous l'avons observé, différents langages parallèles, qu'ils le soient intrinsèquement ou par extension, visent parfois différents niveaux de représentation d'applications. Certains présentent une solution au niveau SMP, d'autres au niveau inter-nœuds. De cette manière, certains langages se retrouvent complémentaires. Dans le cas d'extensions parallèles, il peut alors être envisageable d'en combiner

plusieurs.

C'est ainsi qu'à l'échelle d'une plateforme, on retrouve souvent l'emploi d'une combinaison d'outils parallèles pour décrire une application. Ainsi, le couple MPI/ OpenMP [47] est utilisé depuis des années pour définir des applications pour HPC. Cette combinaison permet de séparer les niveaux d'abstraction. On pourra alors par exemple dans un premier temps employer une vue locale pour représenter l'algorithme censé s'exécuter sur un nœud et sa manière de dialoguer avec les autres à l'aide de MPI. On suppose alors que chaque nœud ne dispose que d'une unique unité d'exécution. Puis, dans un second temps, pour tenir compte du parallélisme interne à un nœud, on pourra ajouter des directives OpenMP pour permettre l'utilisation du système SMP.

OpenCL étant plus récent, le couple OpenCL/MPI reste pour l'instant moins courant. Mais celui-ci se développe de plus en plus pour permettre d'ajouter au système SMP du nœud la possibilité d'exploiter d'autres ressources que les CPUs. Notamment dans le monde des HPCs, qui commence de plus en plus à accueillir ces unités de calcul, comme on l'a vu avec Tianhe-1A. Dans cette situation, l'utilisation du couple OpenCL/MPI, le premier pour gérer les ressources locales à un nœud, et le second pour la communication entre nœuds, devient appropriée.

2.3 Accélérateurs matériels

Dans la section précédente, nous avons étudié le parallélisme entre unités d'exécution et détaillé différentes manières de l'exploiter. Nous nous sommes principalement intéressés aux unités d'exécution logicielles, c'est-à-dire exécutant séquentiellement un code. Ces unités sont donc capables de réaliser une infinité d'algorithmes différents pour peu qu'on leur fournisse le code adéquat.

Les PEs matériels, ou *accélérateurs matériels*, forment une autre catégorie d'unité d'exécution. À la différence des unités logicielles, celles-ci ne sont pas programmables par le biais d'instructions : un accélérateur matériel n'est conçu pour réaliser qu'un seul type de traitement.

Cette spécialisation permet une grande liberté dans la conception des PEs. Comme nous l'avons dit précédemment, il est notamment possible d'exploiter un parallélisme interne pour réaliser la tâche. L'unité peut également embarquer des opérateurs de calculs spécifiques à la tâche, là où une unité logicielle privilégiera des opérateurs génériques communément utilisés.

Le résultat est donc une plus grande efficacité dans le traitement du processus comparé à une implémentation logicielle. Les accélérateurs matériels sont particulièrement utilisés dans les circuits dédiés, comme les ASICs, dont on connait exactement l'utilisation finale qui en est faite.

Mais le gros handicap des unités d'exécution matérielles est longtemps resté leur caractère figé. En effet, pour un système générique, comment prévoir quelle

utilisation exacte sera faite par l'utilisateur, et donc quelles unités d'exécution embarquer ? Néanmoins, ce handicap commence à se dissiper devant la montée en puissance du matériel reconfigurable.

Dans cette section, nous présenterons le principe de la reconfiguration matérielle en 2.3.1, puis nous étudierons les superordinateurs reconfigurables en 2.3.2.

2.3.1 Reconfiguration matérielle

La reconfiguration matérielle consiste à disposer d'un circuit comportant des ressources matérielles génériques. Il s'agit d'une matrice d'éléments programmables (Field-Programmable Gate Array – FPGA) reliés par un réseau lui aussi programmable, tel que présenté sur la figure 2.7.

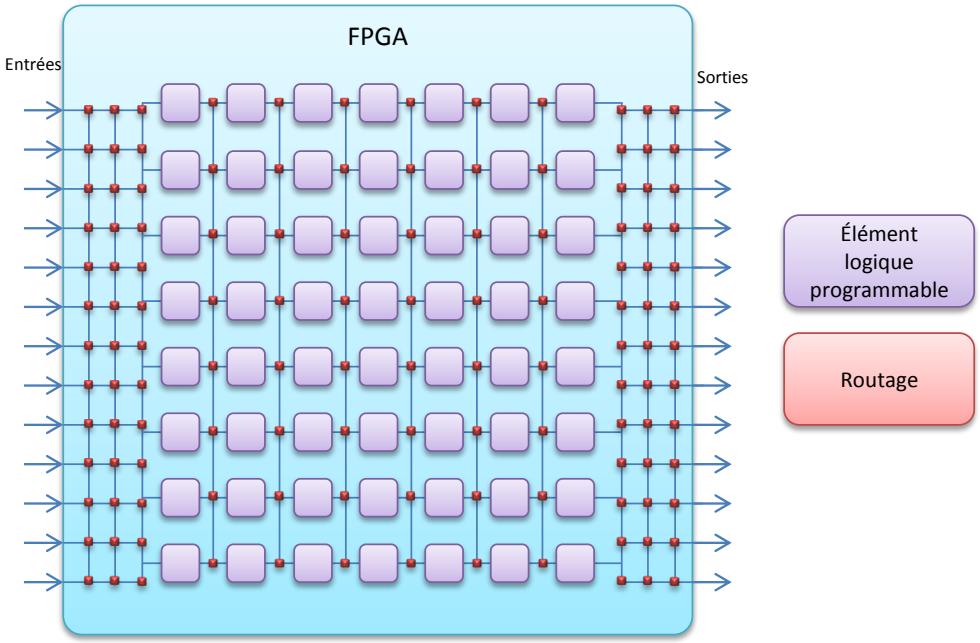


Figure 2.7 – *Structure d'un FPGA.*

Les éléments programmables (Configurable Logic Block – CLB) sont généralement constitués de plusieurs *cellules*. Au sein de chaque cellule, on trouve notamment des tables de correspondance (LookUp Table – LUT), associées à des multiplexeurs et des bascules. Une LUT est une mémoire associant une sortie à une entrée, dans laquelle il est possible d'enregistrer le résultat d'une fonction logique, en associant la valeur souhaitée à chaque combinaison d'entrée. Les LUTs sont associées à des bascules et des multiplexeurs afin de proposer différentes configurations de la cellule.

Le réseau reliant les CLBs est lui aussi programmable, permettant de réaliser un routage des liens logiques afin de réaliser un réseau de CLBs. En combinant la

programmation des CLBs et du réseau les interconnectant, il devient possible de réaliser n'importe quel circuit logique, dans la limite des ressources présentes.

L'utilisation de matériel générique permet donc la constitution dynamique d'unités d'exécution matérielles. Adjoindre un circuit de type FPGA à des unités d'exécution logicielles permet ainsi d'avoir un PE adaptable aux besoins de l'application. Il est également possible de construire un circuit composé de plusieurs unités sur un seul FPGA (System on Programmable Chip – SoPC). Une limitation à ce type de système est que les FPGAs ont longtemps été reconfigurables d'un seul bloc. Ceci est notamment du au fait que l'utilisation d'un FPGA est fortement liée aux outils mis à disposition par le constructeur pour générer les configurations et reconfigurer le dispositif. Disposer la totalité du SoPC sur un seul FPGA limitait donc les possibilités de modification « à chaud », car une reconfiguration interrompait la totalité du système.

Dans ces conditions, l'utilisation de matériel reconfigurable a tout de même un avantage par rapport à du matériel statique : les possibilités d'évolution. En effet, la configuration d'un FPGA est constituée d'un simple fichier enregistrant l'état voulu des éléments internes, et chargé au démarrage du système ou lors d'une opération de reconfiguration. Dans ces conditions, il est donc possible de mettre à jour une configuration matérielle de la même manière que l'on met à jour un programme logiciel, en changeant le fichier de configuration. Cette évolutivité mène donc parfois au choix de l'utilisation de matériel reconfigurable même pour un usage statique.

Néanmoins, les restrictions liées à la reconfiguration en un seul bloc sont elles aussi en train d'être levées grâce à la technique de la reconfiguration partielle.

2.3.1.a Reconfiguration dynamique partielle

La reconfiguration partielle consiste à disposer d'un fichier de configuration ne rassemblant les informations que d'une partie du SoPC. En utilisant ce fichier, on modifie alors seulement une partie du système sans toucher aux autres éléments du SoPC. De plus, la reconfiguration *dynamique* partielle permet d'effectuer cette modification pendant que le reste du SoPC continue à fonctionner normalement, sans avoir à le suspendre.

On utilisera le terme *bloc de propriété intellectuelle* (Intellectual Property block – IP) pour désigner une unité vue comme un ensemble ayant une fonctionnalité particulière. La Reconfiguration Dynamique Partielle (RDP) casse donc le caractère monolithique du FPGA, et permet enfin de l'envisager comme un véritable système composé de plusieurs IPs, et notamment plusieurs unités d'exécution. Birk [8] compare cette possibilité d'instanciation d'IPs au moment voulu au concept logiciel de bibliothèque dynamique. En effet, la RDP rend tout à fait possible d'utiliser plusieurs instances du même IP en reconfigurant plusieurs zones avec ce même IP si besoin.

Dans les FPGAs Xilinx, la reconfiguration partielle peut être contrôlée de l'in-

térieur même du FPGA grâce à l'ICAP (Internal Configuration Access Port) [89]. Néanmoins, celle-ci n'étant pas facile d'utilisation ni très rapide, des surcouches ont été développées pour permettre une interaction simple. Nous nous intéressons particulièrement à FaRM (Fast Reconfiguration Manager) [23, 24]. Ce gestionnaire fait office d'interface entre un bus et l'ICAP, et permet une gestion de la RDP très simplifiée. En effet, à partir d'un fichier de configuration partiel stocké en mémoire, il suffit de donner à FaRM l'adresse et la taille de ce fichier, puis tout le processus de reconfiguration est automatisé. De plus, FaRM supporte la compression des fichiers de configuration, permettant de réduire les temps de transferts du fichier.

2.3.1.b Utilisation de matériel reconfigurable : avantages et compromis

L'utilisation du terme *accélérateur matériel* suggère que l'utilisation de PEs matériels permet une accélération de la vitesse de traitement d'un noyau. Néanmoins, selon le système, la vitesse d'exécution peut ne pas être l'effet recherché. Dans le cas des systèmes embarqués, la consommation peut par exemple être un critère prédominant. Il est alors possible, pour le même PE matériel, de diminuer la fréquence d'horloge. Cela a pour effet de diminuer la vitesse du système, mais également la consommation de celui-ci. On peut donc, à performances équivalentes avec du logiciel, obtenir une consommation plus faible en utilisant une implémentation matérielle [52, 51].

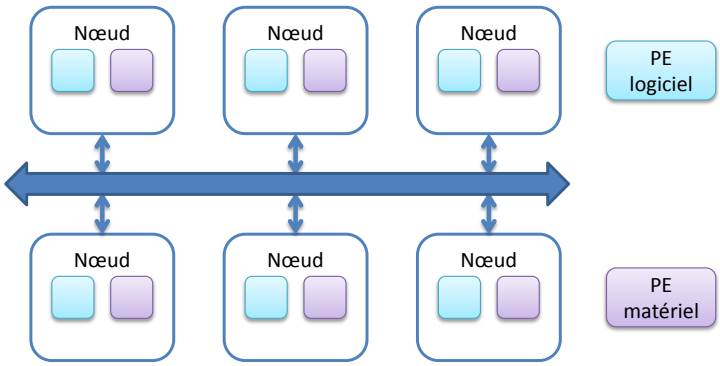
Un autre critère est le taux d'occupation du matériel reconfigurable. En effet, il est parfois possible de proposer différents niveaux de parallélisation pour la structure d'un PE matériel, menant à différents niveaux de performances. Bien entendu, un plus fort taux de parallélisation utilisera plus de ressources reconfigurables, pour permettre d'implémenter les différentes branches parallèles. Il faut donc souvent trouver un compromis entre la vitesse d'exécution, la consommation et le nombre de ressources utilisées lors de la conception d'un PE matériel.

La reconfiguration ouvre donc de nouveaux horizons en termes de puissance de calcul en autorisant l'usage d'unités d'exécution matérielles quelle que soit l'application, et sans avoir à anticiper ces besoins à la construction de la plateforme. C'est donc tout naturellement que le matériel reconfigurable commence à être utilisé dans le milieu du calcul haute performance.

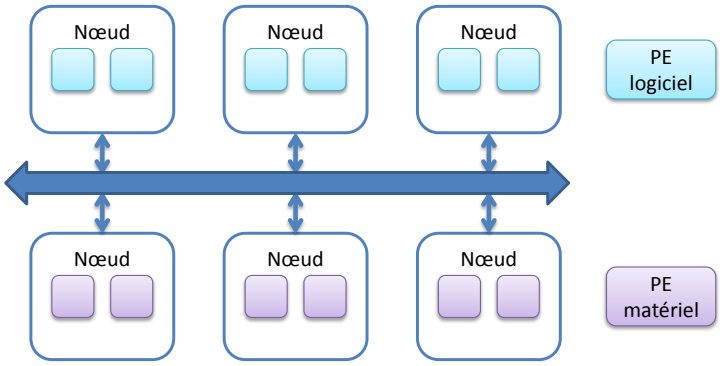
2.3.2 HPRCs

Les HPCs, en intégrant le matériel reconfigurable, changent d'appellation. On parle ainsi de *High Performance Reconfigurable Computers*, ou HPRCs, afin de bien identifier ce domaine d'étude.

Le matériel reconfigurable peut s'intégrer de deux manières dans un superordinateur. El-Ghazawi [26] identifie ces deux modes de conception sous les termes UNNS et NNUS, présentés sur la figure 2.8. Les HPRCs à nœuds hétérogènes



(a) *HPRC de type Nonuniform Node Uniform System*



(b) *HPRC de type Uniform Node Nonuniform System*

Figure 2.8 – *Les deux formes de HPRCs*

constituant un système homogène (Nonuniform Node Uniform System – NNUS) sont constitués de nœuds contenant à la fois des ressources logicielles et matérielles (2.8a). Le système global est donc homogène car tous ses nœuds sont identiques. À l'inverse, dans un HPRC à nœuds homogènes constituant un système hétérogène (Uniform Node Nonuniform System – UNNS), chaque nœud ne contient qu'un seul type de ressource (2.8b). Le système résultant est donc hétérogène car constitué de deux types de nœuds, les nœuds logiciels et les nœuds matériels.

Chaque type de système a son avantage. Il est très facile de transformer un HPC en HPRC UNNS : il suffit d'ajouter des nœuds matériels aux nœuds logiciels existants. On augmente alors la puissance de calcul du système existant sans avoir à repartir de zéro.

Néanmoins, la vitesse du lien existant entre les PEs logiciels et les PEs matériels est souvent déterminante. En effet, dans la plupart des cas, les accélérateurs matériels sont utilisés en tant que *coprocesseurs* adjoints aux PEs logiciels. Une utilisation de type coprocesseur signifie que l'application principale est exécutée sur le PE logiciel, qui fait appel au coprocesseur pour exécuter des instructions spécifiques. Pour les accélérateurs matériels, il s'agit donc d'utiliser ces PEs pour exécuter des calculs particuliers dans l'application à la demande d'un PE logiciel.

À cette fin, il faut alors transférer les données sur lesquelles porte l'exécution entre le PE logiciel et le PE matériel. Une plateforme NNUS aura donc l'avantage de la proximité des PEs, qui peuvent être reliés physiquement par un bus rapide, là où un UNNS devra les faire transiter par le réseau inter-nœuds, plus lent. Un exemple mettant en lumière l'importance de la vitesse de ce lien a été réalisé par Gothandaraman sur une application de type Monte-Carlo [36].

Concernant la RDP, les HPRCs commerciaux actuels ne la supportent pas encore. Celle-ci est toujours au stade de la recherche, notamment du fait de sa complexité en l'état actuel des choses. Néanmoins, de premières applications de la RDP aux systèmes HPRCs ont vu le jour. On citera notamment les travaux d'El-Araby, qui a effectué des tests sur un châssis HPRC commercial [25].

2.4 Interfaces des IPs : communication et interaction

La question de la coexistence logiciel/matériel dans une plateforme impose une définition des interfaces. L'interface est le lien entre une unité d'exécution et le reste de la plateforme. Concernant les PEs logiciels, la question de l'interface ne se pose en général pas. En effet, dans la plupart des systèmes, les PEs logiciels sont considérés comme *maître*, c'est-à-dire qu'ils contrôlent leur environnement et non l'inverse. L'interface est alors standardisée et gérée par le système d'exploitation par le biais d'opérations génériques transparentes pour le concepteur de l'application.

En revanche, les PEs matériels sont en général vus en tant qu'*esclaves*. Le PE matériel est alors contrôlé par un PE logiciel, et vu comme un coprocesseur. Le concepteur du PE définit une interface pour permettre de commander celui-ci. Afin de proposer une interaction simple de la couche logicielle avec le PE, on utilise des *pilotes*. Le pilote est un code logiciel qui permet de transformer les appels d'applications au matériel en actions sur l'interface du PE matériel.

Une manière simple de connecter un PE matériel à une plateforme est de placer celui-ci sur le bus central. Sur ce même bus est donc connecté un ou plusieurs PEs logiciels en tant que maître, qui peuvent alors initier des actions sur l'interface des PEs matériels, esclaves. L'avantage de l'utilisation d'un bus est de réduire les interactions à des lectures/écritures dans des *registres*, c'est-à-dire des zones mémoires adressables. Il est alors possible d'utiliser l'IP quelle que soit la plateforme, simplement en adaptant l'interface au protocole du bus et en écrivant le pilote adéquat.

Si l'utilisation d'un bus permet une généricité de l'interface, on préfèrera parfois définir un protocole spécifique à l'IP. C'est le cas dans certains ASICs, ou la généricité n'est pas nécessaire et où une interface spécifique peut permettre de gagner en vitesse. Par exemple, si deux PEs doivent communiquer directement entre

eux, le recours à un bus n'est pas nécessaire. Certains PEs logiciels définissent une interface spécifique pour brancher un coprocesseur, ce qui permet un canal de communication dédié. Néanmoins, la plupart des PEs logiciels standard utilisant une interface de type bus, leur utilisation en tant que maître rend nécessaire le passage par un bus pour communiquer avec un PE matériel.

Définir des interfaces de communication standard est une chose, décrire ces mêmes interfaces en est une autre. Décrire une interface consiste à lister les signaux dont elle est composée, ainsi que leur type et leurs caractéristiques temporelles. Dans le cas d'un accès par registre, comme un bus, il faudra également définir les plages mémoire disponibles, leur direction, leur taille, etc. IP-XACT [54] est un standard IEEE défini par le consortium SPIRIT. Il s'agit d'une représentation en XML (Extensible Markup Language) destinée entre autres à définir l'interface d'un IP de manière standard. Ainsi, IP-XACT permet de détailler une interface en indiquant pour chaque signal la composant les caractéristiques nécessaires à son utilisation. L'ensemble des informations sur les signaux de l'interface permet ainsi de manipuler celle-ci.

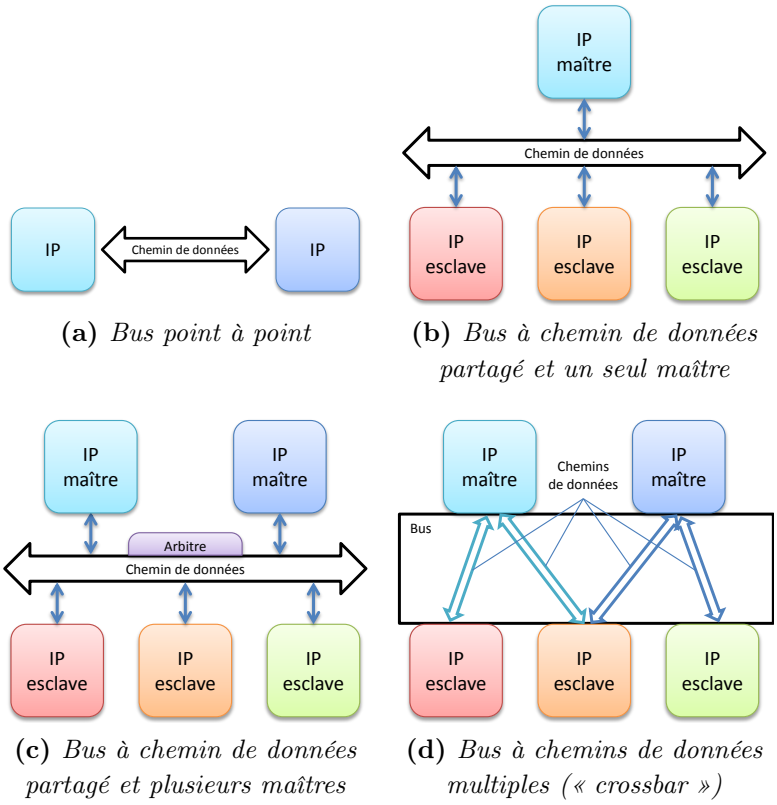
2.4.1 Bus

Le bus est le mode de communication privilégié pour la conception de systèmes sur la base d'IPs, souvent qualifiés de « briques matérielles », que l'on assemble. En effet, afin de réutiliser des IPs existants, l'utilisation d'un bus permet de les brancher très simplement, puis de ne plus avoir à se focaliser sur le protocole de bas niveau, défini par le standard du bus. On réfléchit alors non pas en termes de signaux, mais de *transactions*, consistant en des lectures, des écritures et des requêtes. Utiliser un bus définit donc une couche d'abstraction dont la gestion reste simple.

De nombreux standards de bus existent, que nous ne pourrons tous citer. Nous donnons ici seulement quelques exemples afin de montrer la diversité existante. On présente les topologies de bus les plus communes sur la figure 2.9 [29]. De plus, comme indiqué précédemment, un nouveau type de bus se développe de plus en plus : les NoCs. Bâtis comme des réseaux, ceux-ci sont composés de routeurs permettant de diriger un message vers sa destination. Ceux-ci deviennent utiles lorsque le nombre de PEs commence à croître au-delà d'une dizaine, les accès à un bus standard devenant un véritable goulet d'étranglement. Un exemple de bus de type NoC est présenté sur la figure 2.10, mais différentes topologies sont possibles pour le raccordement des PEs.

AMBA

Advanced Microcontroller Bus Architecture (AMBA) n'est pas à proprement parler un bus, mais une famille de bus, définie par ARM. Les principaux bus de cette



(a) *Bus point à point*

(b) *Bus à chemin de données partagé et un seul maître*

(c) *Bus à chemin de données partagé et plusieurs maîtres*

(d) *Bus à chemins de données multiples (« crossbar »)*

FIGURE 2.9 – *Les principaux types de bus.*

famille sont Advanced eXtensible Interface (AXI), Advanced High-performance Bus (AHB) et Advanced Peripheral Bus (APB).

APB [1] est un bus destiné à interconnecter des IPs à faible débit, c'est-à-dire n'ayant que peu de données à transférer, par exemple des IPs de type clavier ou UART (Universal Asynchronous Receiver Transmitter – Émetteur-récepteur asynchrone universel, gérant les liaisons de type série). APB permet un ensemble d'opérations de lecture/écriture très simple et très réduit, avec un seul maître par bus. Notamment, ce bus ne permet que des opérations unitaires, c'est-à-dire de transférer un seul mot de donnée par requête de transfert.

Dès que l'IP devient plus gourmand en bande passante, APB et ses spécifications réduites deviennent un facteur limitant. Notamment, le mode *rafale* (« burst »), qui permet de transférer plusieurs mots à la suite après une seule opération de requête, n'est pas présent dans APB. Pour ce type de besoins, on se tourne plutôt vers AHB [1]. Il s'agit d'un bus permettant des lectures/écritures de différentes tailles et supportant le mode rafale, ainsi que plusieurs maîtres. Les différents maîtres étant en concurrence pour l'accès au bus, celui-ci dispose d'un *arbitre* pour distribuer l'accès. Il est possible d'effectuer des opérations sur 4, 8 ou 16 mots successifs, mais également sur des tailles de burst non spécifiées, qui continueront tant que l'IP ayant fait la requête en a besoin, ou que l'arbitre de bus décide de donner

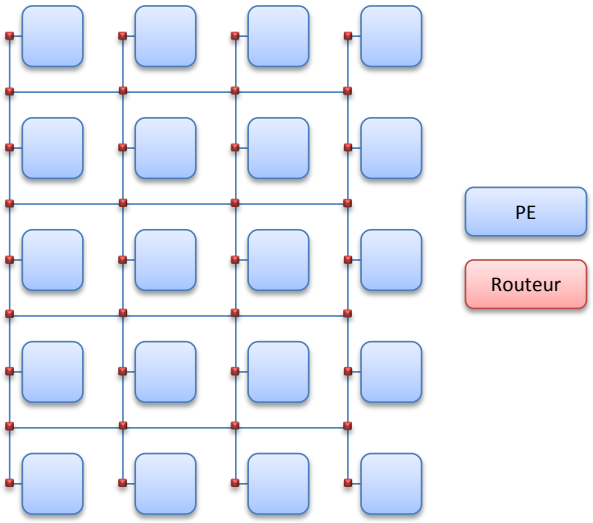


Figure 2.10 – *Exemple de structure d'un NoC.*

l'accès à un autre IP. Ce bus dispose également de signaux d'erreur permettant aux esclaves de signifier au maître ayant initié la transaction qu'une opération n'a pas pu aboutir.

Ces deux types de bus sont relativement conventionnels, chacun ayant son intérêt selon le type d'IPs à relier. Néanmoins, comme on l'a vu, le nombre de PEs dans un système est en constante augmentation. Selon les cas, l'accès à un seul bus, central, pour tous les IPs, peut constituer un goulet d'étranglement limitant fortement les performances d'un système. On se tourne alors vers des solutions de type NoC.

C'est ainsi que AXI [2], dernier né de la famille AMBA, n'est pas centré sur une interconnexion partagée en particulier. AXI propose un standard concernant l'interface entre un maître et une interconnexion, l'interconnexion et un esclave, ainsi qu'en point à point entre un maître et un esclave. En revanche, le standard ne définit pas la topologie de l'interconnexion elle-même, laissant le champ libre pour le choix de l'implémentation, comme présenté sur la figure 2.11. Il est ainsi possible d'utiliser l'interface AXI sur un bus classique, dont le chemin de données est partagé, mais également sur d'autres types de bus. Par exemple, il est possible de réaliser un bus de type « crossbar », dans lequel chaque maître dispose de sa propre connexion aux esclaves, évitant la concurrence sur le chemin de données, et permettant ainsi plusieurs communications maître/esclave simultanées pour peu que les maîtres *et* les esclaves soient différents. Enfin, cette liberté permet également d'utiliser un vrai NoC basé sur des échanges de messages.

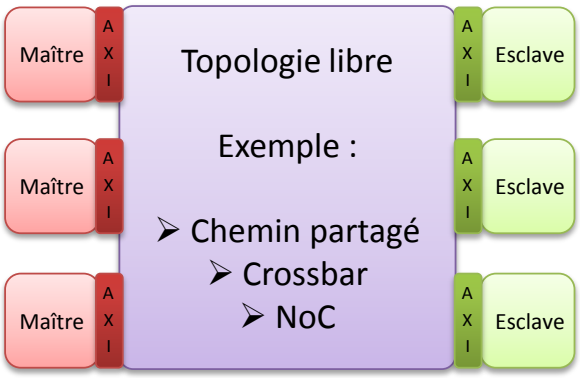


Figure 2.11 – *Principe d'AXI.*

PLB

Processor Local Bus (PLB) [45] fait partie de CoreConnect, une famille de bus IBM, servant notamment d'interface au processeur PowerPC. Le PowerPC est très répandu dans le monde de l'embarqué, et a longtemps servi de base aux processeurs embarqués dans les FPGAs Xilinx. Xilinx a remplacé ceux-ci depuis peu par la famille de processeur ARM Cortex [27], utilisant des bus AMBA, mais PLB reste néanmoins présent dans les architectures bâties sur la base de PowerPC.

PLB est un bus disposant de plusieurs implémentations, permettant de faire varier la taille de l'adresse et du chemin de données. Il permet bien entendu les échanges de type rafale, largement utilisés pour les échanges CPU–Mémoire.

2.4.2 Protocoles

Si la définition de ces standards de bus permet une grande standardisation des accès aux IPs matériels, il peut parfois être utile de définir d'autres types de communication. On trouve ainsi des spécifications qui ne sont pas basées sur un bus particulier, mais permettent une plus grande liberté de choix.

Nous présentons ici des protocoles visant différents niveaux d'interactions afin d'effectuer un tour d'horizon des possibilités existantes. Ainsi l'interface de CoreLib se concentre sur les interactions entre IPs à l'aide de signaux dédiés. À l'inverse, OCP présente lui des interfaces, mais sans définir le protocole entre les interfaces. Enfin, Wishbone propose un protocole, mais sans implémentation.

OCP

Open Core Protocol (OCP) [58] est une spécification définie par OCP International Partnership (OCP-IP). OCP est défini comme une interface de type bus, mettant à disposition un jeu de signaux permettant de réaliser des transactions. En revanche, aucun protocole ni aucune topologie n'est indiquée pour connecter

ces interfaces, comme indiqué sur la figure 2.12. Par exemple, lorsqu'un maître ef-

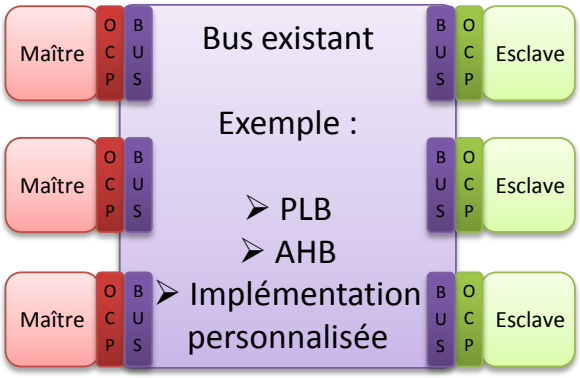


Figure 2.12 – *Principe d'OCP.*

fectue une transaction sur l'interface OCP correspondante, cette action doit être répercutée sur l'interface esclave cible, mais aucune contrainte n'est donnée sur la manière de propager les signaux entre ces deux interfaces. On peut donc se baser sur n'importe quel type de bus pour transmettre les données entre les interfaces, en « enveloppant » celui-ci d'une couche supérieure correspondant au protocole OCP.

L'intérêt de cette indépendance se trouve dans la possibilité de réutiliser des IPs. En effet, quel que soit le bus réel existant sur une plateforme, la communication via OCP permet l'utilisation d'une interface qui, elle, est commune. Ainsi, utiliser un IP sur une nouvelle plateforme ne requiert aucun changement, l'interface de communication ne changeant pas. Il suffit de disposer d'une couche de compatibilité entre OCP et le bus de la plateforme pour pouvoir réutiliser n'importe quel IP.

OCP permet les interactions basiques de type lecture/écriture, y compris en mode rafale. Un mode de diffusion un-vers-plusieurs (« broadcast »), permettant d'envoyer une donnée vers plusieurs périphériques simultanément, est également supporté. Enfin, un autre ajout est la lecture de type bloquée : une lecture est effectuée, suivie d'une écriture, sans qu'aucun autre périphérique ne soit autorisé à utiliser le réseau de communication entre ces deux opérations. Cela permet principalement l'utilisation de l'exclusion mutuelle, en permettant à un périphérique de verrouiller l'accès au bus durant le processus « lecture-analyse-écriture ».

OCP est utilisé dans l'industrie pour la définition d'IPs réutilisables et interconnectables. Ainsi, par exemple, la plateforme OMAP de Texas Instruments, embarquant une architecture ARM, est compatible avec OCP [9]. Néanmoins, ce protocole n'est pas officiellement supporté par les deux principaux concepteurs de FPGAs, Altera et Xilinx.

CoreLib

CoreLib [87], un projet développant un ensemble de standards pour la conception d'IPs matérielles, propose une standardisation d'interface destinée à permettre d'interagir avec eux via des primitives pouvant être appelées sans en connaitre les détails. L'idée est de permettre la création de bibliothèques d'IPs immédiatement utilisables. Le standard défini par CoreLib se focalise sur la génération de circuits personnalisés, et donc n'est pas centré sur un bus, mais se veut adapté à tous types d'interfaces personnalisées, incluant les bus mais également la communication point à point, le multiplexage, etc. Pour cela, le modèle décrit l'interface en trois éléments : la structure, la temporalité et le contrôle, définis à l'aide de la description IP-XACT.

La structure consiste à identifier tous les signaux composant l'interface, leur direction, leur catégorie et leur type. Ainsi, la catégorie permet de définir des signaux d'horloge, de reset, de contrôle et de données. Le type, quant à lui, correspond à la nature du signal transporté : entier signé ou non, nombre décimal, booléen, etc.

La temporalité permet d'indiquer les intervalles à respecter sur un signal, d'identifier les éventuels pipelines et leur profondeur, etc. afin de permettre la détermination de motifs valides pour les signaux.

Enfin, le contrôle est destiné à décrire la manière dont l'IP communique, c'est-à-dire par exemple comment démarrer un calcul ou savoir que celui-ci est terminé.

Contrairement à OCP, CoreLib est pour l'instant plus un effort de recherche qu'une spécification réellement utilisée dans l'industrie.

Wishbone

Soutenue par OpenCores [60], Wishbone [65] est une spécification libre et gratuite. L'un des objectifs de cette description est de mettre dans le domaine public des technologies de communication pour empêcher un verrouillage du secteur sous la forme de brevets.

Celle-ci définit de manière logique différents signaux composant l'interface d'un IP, sans préciser de réelle implémentation pour celle-ci. Ainsi, un signal sera défini selon l'état logique « haut » ou « bas », qui peut correspondre selon l'implémentation à un 0 ou un 1.

Cette absence de précision est destinée à permettre l'interfaçage de plusieurs IPs écrits dans un langage de description matériel. En effet, les IPs reconfigurables sont conçus à l'aide de langages de description, qui sont synthétisés sous la forme de configurations pour des composants reconfigurable. Les IPs statiques utilisent également souvent cette description dans un premier temps, avant de transformer la description en circuit logique puis de concevoir un design optimisé. Néanmoins, cette spécification cible avant tout les circuits logiques de type System on Chip

(SoC), contenant des connections courtes, contrairement aux bus destinés à l'interconnexion dans des plateformes plus lourdes.

Il est ainsi possible de décider comment sont implémentés ces signaux en fonction des IPs utilisés. L'interconnexion entre les interfaces offre différentes topologies, notamment le point à point, le bus partagé et le crossbar.

2.5 Plateformes existantes

La recherche sur le matériel reconfigurable a été très active ces dernières années, et de nombreuses plateformes ont vu le jour. Nous en présentons ici quelques unes qui nous semblent représentatives de l'effort de recherche.

Æther

Le projet européen Æther [71] a travaillé sur une plateforme composée d'ensembles d'éléments reconfigurables interconnectés [10]. Chaque élément de base, nommé cell (cellule), est reconfigurable et exprime une fonction de base du système. On choisira la granularité de la cellule selon le système, pouvant varier d'une porte logique à un processeur complet. Les cellules adjacentes peuvent être rassemblées pour former des composants, pouvant eux-mêmes être rassemblés sous la forme de SANEs (Self-Adaptive Network Element). Enfin, les SANEs sont mis en réseaux pour constituer le système complet. La reconfiguration des cellules combinée à la possibilité de changer la composition des composants pour inclure un nombre variable de cellules permet une grande flexibilité dans la conception d'un système dynamique.

BORPH

Berkeley Operating system for ReProgrammable Hardware (BORPH) [79] est un système d'exploitation utilisant une interface UNIX pour les ressources matérielles. L'interface UNIX étant un standard logiciel, son extension aux PEs matériels permet une compatibilité avec les applications UNIX, prépondérantes dans l'écosystème numérique. Les éléments matériels sont représentés par des processus, au sens UNIX du terme, au même titre que des processus logiciels, et donc gérés de la même manière par le système d'exploitation. La compatibilité de l'interface avec UNIX permet notamment une communication native avec les processus logiciels, par exemple en utilisant le *pipe* dans une console. Conçu comme une surcouche de Linux, BORPH permet donc une intégration transparente des PEs implémentés sur matériel reconfigurable au sein d'un environnement éprouvé.

Egret

La plateforme Egret (Experimental Generic Reconfigurable Embedded Target) [7] est composée d'éléments modulaires se connectant physiquement les uns aux autres. Chaque module apporte une fonctionnalité spécifique à la plateforme : alimentation en énergie, processeur, mémoire ou encore connecteur réseau. Sur la base de cette plateforme, les auteurs décrivent les avantages de l'utilisation d'un système d'exploitation embarqué pour la gestion de la reconfiguration dynamique [86]. En l'occurrence, ils utilisent uClinux sur un processeur MicroBlaze. La reconfiguration est autorisée directement depuis la console, en transférant un fichier de configuration vers le pilote de l'ICAP. Le FPGA contient un SoC reconfigurable, permettant une modularité dans le nombre de périphériques présents. Ainsi, les périphériques disposent d'une interface au sein du FPGA, les accès à cette interface étant redirigés sur les ports physiques auquels le périphérique est connecté.

ReMAP

ReMAP (Reconfigurable Multicore Acceleration and Parallelization) [84] est une architecture basée sur des PEs logiciels adossés à des éléments de logique reconfigurable. Les PEs logiciels sont « noyés » dans un réseau reconfigurable permettant de mettre en place dynamiquement des liens de communication entre les unités d'exécution. Ce réseau reconfigurable offre ainsi une communication dynamique, permettant de relier à la demande les différents PEs en fonction des besoins. De plus, les éléments reconfigurables disposent de capacités de calcul, ce qui permet ainsi différents scénarios. Tout d'abord, la simple communication : le PE A communique avec le PE B. Le réseau est alors reconfiguré en tant que simple lien transférant les données de A vers B. Seconde possibilité, combiner communications et calculs. Différentes opérations sont alors possibles au cours de la communication. Une utilisation concerne les opérations de réduction. Par exemple, si les données provenant du PE A et du PE B doivent normalement être récupérées par le PE C, qui en fera la somme. Il est alors possible d'implémenter cette somme directement au cours de la communication. Le PE C reçoit alors le résultat, et n'a pas à effectuer le calcul. Enfin, une troisième possibilité d'utilisation de la logique reconfigurable consiste à utiliser celle-ci en tant que coprocesseur. Il s'agit alors d'un cas particulier du second scénario, dans lequel le PE source et le PE cible sont les mêmes. Cette possibilité permet ainsi de déléguer certains calculs à des opérateurs matériels.

OS4RS

En plus de la plateforme en elle-même, le système d'exploitation doit permettre de gérer celle-ci. Operating System for Reconfigurable Systems (OS4RS) [53] est

un système d'exploitation destiné à rassembler en une seule plateforme différents PEs. Il peut par exemple accueillir des DSPs comme de la logique reconfigurable. Les éléments de base, nommés tiles (tuiles), sont interconnectés par un NoC. L'idée de base est d'offrir un système d'exploitation permettant l'usage de ces ressources hétérogènes de manière transparente, en ajoutant une couche à un système d'exploitation existant. Le système est centré sur un processeur, exécutant l'OS, et décidant de l'ordonnancement des tâches sur les différents PEs. La communication passant par le NoC commun, les tâches peuvent communiquer directement sans passer par l'OS central.

Les MPSoCs reconfigurables

La notion de SoC multiprocesseur (MultiProcessor SoC – MPSoC) reconfigurable [5, 91] consiste à placer un environnement multiprocesseur sur plateforme reconfigurable. Il s'agit donc ici de plateformes principalement logicielles, les applications s'exécutant sur des processeurs, parfois épaulés par des accélérateurs matériels. Une première reconfiguration matérielle s'effectue en amont de l'exécution d'applications, afin de mettre en place les PEs, notamment logiciels. Cette première phase permet de disposer les cœurs d'exécution logiciels de manière adaptée à l'application. Par la suite, différentes reconfigurations permettent de disposer des accélérateurs matériels, destinés à servir de coprocesseurs, accélérant ainsi l'exécution.

2.6 Ordonnancement

L'ordonnancement des tâches est un problème complexe, particulièrement dans les systèmes hétérogènes. Dans le cas où plusieurs implémentations existent pour des ressources de natures différentes, le choix est parfois ardu. Il faut définir différents critères selon des métriques qui ne se calculent pas forcément de la même manière selon la nature du PE. Nous présentons dans cette section des recherches effectuées sur l'ordonnancement hétérogène.

Application Heartbeats

L'environnement d'exécution d'applications hétérogènes Application Heartbeats propose une API de surveillance (« monitoring ») des applications [67]. Chaque application peut disposer de plusieurs implémentations, et les performances de celles-ci sont mesurées en fonction du *rythme cardiaque* (« heartbeats »). Le rythme cardiaque est une unité de mesure, consistant à faire remonter à l'environnement un « battement cardiaque » lors de la fin du traitement d'un bloc de données. L'objectif d'exécution de l'application est défini en choisissant un rythme cible, et

l'implémentation de l'application est dynamiquement changée de manière à s'approcher au mieux de cet objectif.

L'API de monitoring est une approche intéressante du choix d'ordonnancement en ligne, en permettant de réaliser le choix d'une implémentation d'une fonctionnalité en fonction d'heuristiques. Le changement d'une implémentation à une autre est autorisé par l'utilisation d'un traducteur de structures de données, qui transforme la structure des données utilisées par une implémentation de manière à ce que celles-ci soient utilisables par la nouvelle implémentation choisie. Néanmoins, ce traducteur doit être fourni pour chaque implémentation d'une fonctionnalité, et est donc spécifique à une application, nécessitant un développement supplémentaire, et demandant de connaître la représentation interne des données par l'implémentation [68].

AMAP

AMAP, pour *adaptive mapping algorithm* [66], est un module de décision permettant de choisir à l'exécution entre une implémentation logicielle et une implémentation matérielle. L'objectif de cet algorithme est de prendre en compte tous les délais existants lors d'une exécution : temps de reconfiguration matérielle, temps passé à transférer les paramètres, etc. Sur cette base, pour une tâche disposant des deux implémentations, le but est de savoir avant l'exécution quelle implémentation sera la plus rapide, en fonction de la taille des paramètres. En effet, pour une quantité importante de données à traiter le temps de reconfiguration peut être négligeable et favoriser une exécution matérielle. En revanche, il est possible que pour traiter une faible quantité de données, l'exécution logicielle soit plus rapide, car le délai introduit par temps de reconfiguration est plus long que le temps d'exécution logiciel.

Sur ce constat, AMAP construit un arbre contenant les différentes combinaisons de tailles de paramètres utilisées, et y associe les temps d'exécution totaux pour chaque implémentation. Ainsi, lors de l'exécution, il suffit de rechercher dans l'arbre la combinaison courante des tailles de paramètres pour retrouver les temps d'exécution des dernières exécutions avec ces paramètres. Ceci permet alors de choisir l'implémentation la plus rapide directement à l'exécution.

2.7 Conclusion

Dans ce chapitre, nous avons fait un tour d'horizon de l'état de l'art dans les domaines concernés par notre étude après en avoir défini les termes importants. Nous nous sommes intéressés au parallélisme, dans lequel nous avons identifié plusieurs niveaux : parallélisme interne au PE, mise en parallèle de PEs sur une mémoire partagée, puis mise en parallèle de systèmes à mémoire partagée. Cette dernière

catégorie, qui concerne principalement les HPCs, résiste facilement au passage à l'échelle, permettant de proposer des supercalculateurs contenant plusieurs dizaines de milliers de PEs. Puis, nous avons détaillé le principe du matériel reconfigurable, qui consiste à proposer un circuit logique générique permettant d'instancier différents blocs logiques, accélérateurs matériels, processeurs, mémoires, etc. Nous avons également mis en avant les interfaces permettant d'articuler les communications entre les PEs. Enfin, nous avons détaillé quelques plateformes reconfigurables existantes, ainsi que des travaux concernant l'ordonnancement. Sur cette base, nous sommes maintenant capables de nous focaliser sur notre problématique, et de nous situer par rapport l'existant.

Chapitre 3

Spécification du modèle d'application et de la plateforme

Sommaire

L'objectif de ma thèse est de proposer une solution de prise en charge simple et transparente de la gestion du matériel afin de permettre la flexibilité des applications hétérogènes de traitement de données parallèles. Nous avons constaté dans l'état de l'art qu'un moyen pour combiner performances et flexibilité consistait en l'utilisation de la reconfiguration matérielle partielle. Nous nous intéressons donc aux applications parallèles déployées sur plateforme supportant cette technologie. Notre but est de définir un modèle de description pour le type d'application visé ainsi qu'une structure de plateforme compatible, avec pour objectif la simplicité d'utilisation des PEs logiciels et matériels, particulièrement reconfigurables.

Sur la partie applicative, il convient de déterminer le profil des applications auxquelles on s'intéresse, en déterminant les critères pertinents. Une fois défini le profil d'application auquel on se réfère, une méthode de modélisation adaptée à leur structure sera proposée ainsi qu'un modèle graphique permettant de représenter les applications parallèles.

En ce qui concerne la plateforme, il s'agit de déterminer un environnement permettant d'accueillir un déploiement d'applications exploitant ce modèle. À cette fin, nous proposerons une structure suffisamment générique pour pouvoir utiliser différents types d'unités d'exécutions.

Sur cette base, nous définissons dans ce chapitre un système théorique à même de répondre aux différents besoins identifiés en termes de modèle d'application et de plateforme. Ce système théorique sera à la base d'implémentations détaillées dans les chapitres 4 et 5.

3.1 Analyse et modélisation du profil d'application

Dans notre étude, il existe deux types d'unités d'exécutions : les PEs logiciels et les PEs matériels. Ceux-ci peuvent être présents nativement dans une plateforme, ou bien implémentés dynamiquement sur du matériel reconfigurable. L'un de nos objectifs concerne le support d'exécution par les applications de différentes plateformes, ajoutant ainsi une contrainte de portabilité. De plus, la contrainte d'hétérogénéité peut s'exprimer tant entre deux plateformes qu'au sein d'une seule plateforme. Par exemple, les processeurs peuvent utiliser différents jeux d'instructions et les ressources reconfigurables de technologies différentes. Il est donc nécessaire de prendre en compte cette forte hétérogénéité pouvant exister dans les systèmes de traitement lors de la conception des applications.

Une application est composée de deux éléments : la partie contrôle et la partie traitement des données. La partie traitement est représentée par l'ensemble des algorithmes mis en œuvre pour effectuer des opérations sur des données. Le propre d'une application parallèle est de permettre l'exécution de plusieurs processus simultanément. On identifie ceux-ci sous le nom de *noyaux* de calcul, qui s'exécutent sur différents PEs et peuvent communiquer entre eux. Cette partie dépend direc-

tement de la plateforme, vu que les noyaux sont représentés par des exécutables et/ou des fichiers de configuration dépendant respectivement d'un jeu d'instruction logiciel ou d'une technologie de matériel reconfigurable.

La partie contrôle est gérée par l'environnement et détermine quels noyaux doivent être exécutés à un instant donné. Celle-ci n'agit pas directement sur des données, mais peut éventuellement être influencée par des données de configuration. La partie contrôle peut être représentée de manière indépendante de la plateforme, en n'utilisant pas un jeu d'instruction particulier mais une syntaxe indépendante.

Afin de modéliser une application de manière indépendante d'une plateforme, nous opérerons cette distinction entre contrôle et traitement. Une application parallèle est donc représentée par un graphe de noyaux liés par des échanges de données, et coordonnés par du contrôle.

3.1.1 Modèle d'application – Partie contrôle

La partie contrôle représente le squelette de l'application, gérant les flux de données entre noyaux et décidant de leur exécution. Le contrôle est en charge de la décision des noyaux à exécuter à un instant donné, notamment en fonction de différents *tests conditionnels*. La partie contrôle peut ainsi lancer séquentiellement plusieurs noyaux, exécutant le suivant lorsque le précédent a terminé son traitement. Dans le cadre d'une application parallèle, c'est le contrôle qui identifie les noyaux pouvant être exécutés simultanément. Le contrôle permet également d'exécuter un même noyau plusieurs fois successivement dans le cadre de boucles, qui peuvent éventuellement êtres *déroulées* pour permettre à ces exécutions d'être simultanées.

Nous utiliserons le terme *descripteur* pour représenter un ensemble de caractéristiques associées à un élément de l'application. Ainsi, un descripteur d'application définit des actions de contrôle sur des noyaux, ainsi que des opérations de communication de données entre eux.

3.1.1.a Noyaux et vecteurs

Lors de la modélisation d'une application, il est possible de hiérarchiser son architecture en considérant un de ses sous-ensembles comme un noyau. Prenons l'exemple d'une portion d'application comprenant un choix entre plusieurs noyaux à exécuter en fonction du résultat d'un calcul précédent. Dans ce cas, l'ensemble de ces noyaux, combiné au contrôle correspondant à ce choix, est une sous-application qui peut elle-même être représentée par un noyau. Il s'agit donc d'un noyau *virtuel*, c'est-à-dire n'étant pas réellement un traitement de données, car celui-ci dépend en réalité du choix défini dans cette sous-application.

L'application dispose alors de plusieurs niveaux de hiérarchie. À chaque niveau, les noyaux sont représentés par des boîtes noires. Un niveau d'application

ne connaitra alors que les liens de données entre les noyaux, et éventuellement quelques paramètres fonctionnels, tel qu'une échéance dans le cas d'une application temps réel. En revanche, le contenu de chaque noyau n'est pas connu, et peut être représenté par un traitement de données ou une sous-application hiérarchique.

Modéliser une application hiérarchiquement permet un choix dans la granularité d'un noyau lors de son implémentation. En effet, il est alors possible, pour chaque noyau, de le représenter comme une sous-application ou de l'implémenter en tant que traitement. Un noyau qui n'est pas une sous-application mais fait appel aux ressources de traitement est appelé *noyau final*. En hiérarchisant une application au maximum, chaque noyau final se réduit à une action atomique, toute la partie contrôle étant isolée sous la forme d'un graphe dont les nœuds sont les noyaux finaux.

Un noyau final en cours d'exécution, reposant sur une implémentation particulière, est un processus. Plusieurs processus exécutés simultanément peuvent être basés sur un même noyau, dans le cas du parallélisme de données. Le modèle graphique du noyau est représenté sur la figure 3.1.

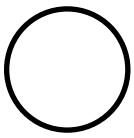


Figure 3.1 – *Représentation graphique d'un noyau.*

Afin de permettre l'expression de haut degré de parallélisme sans augmenter la taille de la représentation d'une application, on définit un vecteur de noyaux pour représenter le parallélisme de données, comme présenté sur la figure 3.2.

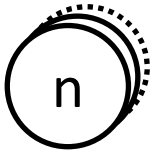


Figure 3.2 – *Représentation graphique d'un vecteur de* n *noyaux.*

Le terme vecteur sera employé par la suite dans son sens large, c'est-à-dire pouvant se résumer à un noyau unique.

3.1.1.b Relations

Dans une application séquentielle, les noyaux sont liés par des *relations de contrôle*, afin de choisir quel noyau doit être exécuté à la suite d'un autre. Les *relations de données* se résument alors aux *dépendances de données*, existant entre

deux noyaux quand l'un d'eux a besoin de données produites par un autre pour pouvoir démarrer.

En revanche, une application parallèle introduit un deuxième type de relation de données. Une *communication* peut être initiée par un noyau source à destination d'un noyau cible. En général, une cible interrompra son exécution en attente des données et restera inactive tant que celles-ci ne seront pas produites par leur source. La communication nécessite donc que les noyaux soient exécutés simultanément pour que l'échange de données fonctionne.

Dans le cas des applications parallèles, nous pouvons concevoir la dépendance de données comme une forme de communication particulière. Néanmoins, la particularité de la dépendance de données réside dans le fait que les données doivent être stockées entre la fin du noyau source et l'exécution du noyau dépendant, et sont donc gérées par l'environnement et non directement par les noyaux.

On utilise la relation de données séquentielle pour identifier une dépendance de données ou une communication entre noyaux, quand la relation parallèle correspond à un échange entre vecteurs. La relation de contrôle séquentielle signifie une exécution successive de noyaux par le biais du contrôle, sans nécessiter de dépendance de données. La relation de contrôle parallèle permet de lancer l'exécution d'un ou plusieurs vecteurs.

Les relations entre les vecteurs seront notées à l'aide de flèches. Il conviendra de différencier le fait qu'il s'agit d'une relation de donnée ou de contrôle. Par ailleurs, il faut également distinguer les relations séquentielles des relations parallèles, où les noyaux s'exécutent simultanément. Pour cela, on différenciera le graphisme des flèches en doublant soit le trait, soit la pointe. Les différentes variations du symbole sont présentées dans le tableau 3.1.

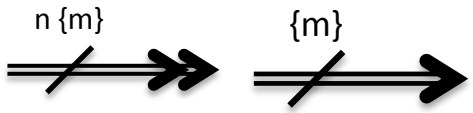
Relation	Contrôle	Données
Séquentielle		
Parallèle		

Tableau 3.1 – *Représentations graphiques des différents types de relations entre noyaux.*

Dans le cas où une relation de contrôle parallèle issue d'un vecteur donne naissance à du parallélisme d'instruction, représenté par plusieurs vecteurs distincts, il y aura plusieurs flèches provenant du vecteur source.

Pour une relation de données, il peut être utile de connaître son dimensionnement. À cette fin, on définit un symbolisme permettant d'indiquer sur le modèle la quantité de données échangées lors d'une relation de données. Comme représenté

sur la figure 3.3, on utilisera pour cela une barre surmontée d'une ou deux valeurs selon le type de relation.



(a) *Relation parallèle dimensionnée*

(b) *Relation séquentielle dimensionnée*

Figure 3.3 – *Représentation graphique du dimensionnement des relations de données.*

Une relation de données séquentielle comporte simplement une taille de données m. Les relations parallèles ajoutent un paramètre n, indiquant le nombre de communications de taille m simultanées. La grandeur m sera exprimée dans une unité arbitraire définie pour l'ensemble du modèle selon la précision souhaitée.

3.1.1.c Rebouclage et pipeline

Des communications peuvent avoir lieu au sein d'un vecteur de noyaux. On symbolise cela à l'aide d'un rebouclage du symbole de communication de données parallèles sur le vecteur, de la manière décrite sur la figure 3.4.

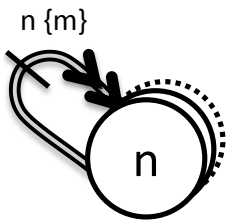


Figure 3.4 – *Représentation graphique d'une communication parallèle entre noyaux d'un vecteur.*

Par ailleurs, il peut arriver qu'un ensemble de noyaux similaires forme un pipeline, les données passant successivement par les éléments du vecteur. Dans ce cas, on utilisera la relation de données séquentielle pour identifier ce cas, comme présenté sur la figure 3.5.

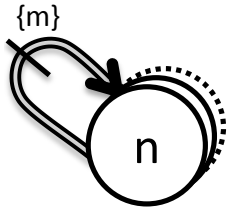


Figure 3.5 – *Représentation graphique d'une communication de type pipeline entre noyaux d'un vecteur.*

3.1.1.d Tests conditionnels

L'expression du contrôle doit permettre d'identifier les tests conditionnels et les boucles. En premier lieu, la relation *si* sert à effectuer des tests afin de prendre une décision. On utilisera une représentation similaire à celle des algorigrammes, utilisant le losange comme représenté sur la figure 3.6

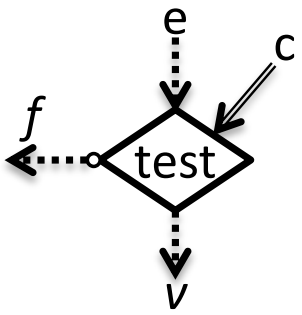


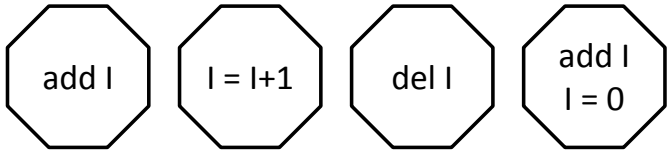
Figure 3.6 – *Représentation graphique d'un test de décision.*

Le contenu du losange représentera le test, par exemple en utilisant la relation de test d'égalité $a \stackrel{?}{=} b$, ou d'inégalité $a \stackrel{?}{\neq} b$. Les pointillés sur la figure 3.6 expriment un type de relation générique. Un test peut donc être appliqué sur tout type de relation, permettant un choix entre deux branches. La relation d'entrée e suivra alors la branche v si *test* est vrai, ou f autrement. On identifie la branche f par un cercle sur la base de la flèche. Toujours afin de dissocier le contrôle du traitement, on ajoute une branche entrante c, qui permet d'amener une donnée extérieure pouvant participer dans l'expression du test indépendamment de la relation d'entrée.

3.1.1.e Variables de contrôle

Afin de définir des boucles, on mettra une nouvelle fois en avant la séparation du contrôle et du traitement en isolant l'*itérateur*. En effet, la variable sur laquelle s'effectue le test du nombre d'itérations, ou itérateur, est un composant de la partie contrôle de l'application.

On définit donc un signe particulier, l'octogone, permettant de définir, mettre à jour, puis finalement supprimer des variables de contrôle. La figure 3.7 présente les différentes actions pouvant s'effectuer sur une variable de contrôle.



(a) *Déclaration* **(b)** *Affectation* **(c)** *Suppression* **(d)** *Déclaration suivie d'une affectation*

Figure 3.7 – *Représentation graphiques des différentes opérations sur des variables de contrôle.*

3.1.1.f Boucles

Le test de décision permet la réalisation de boucles en itérant sur un noyau tant qu'une condition est ou n'est pas vérifiée. Les principaux types de boucles sont les boucles *tant que*, *pour* et *faire ... tant que*. Une boucle *tant que* exécute un noyau tant qu'une condition est vraie, et sort de la boucle lorsque la condition devient fausse. La boucle *pour*, quant à elle, exécute le noyau un nombre déterminé de fois. Enfin, la boucle *faire ... tant que* a le même comportement que la boucle tant que, à l'exception du fait qu'elle est réalisée au moins une fois, même si le nombre d'itérations est nul.

Une boucle *faire ... tant que* est exprimée par la combinaison de symboles de la figure 3.8.

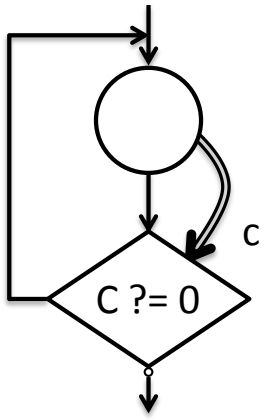


Figure 3.8 – *Représentation graphique d'une boucle* faire ... tant que.

Une boucle *tant que* est exprimée par la combinaison de symboles de la figure 3.9.

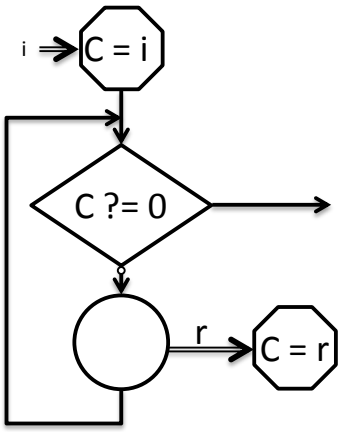
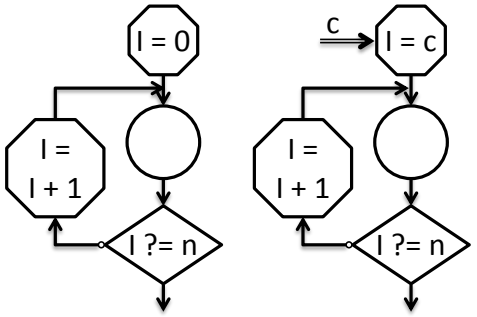


Figure 3.9 – *Représentation graphique d'une boucle* tant que.

À l'aide des variables de contrôle, on peut définir un itérateur qui nous permettra de réaliser une boucle *pour*. La taille des boucles *pour* peut être définie de manière statique, comme représenté sur la figure 3.10a, ou bien déterminée à la suite d'un calcul précédent, comme sur la figure 3.10b.



(a) *Boucle* pour *de taille statique* **(b)** *Boucle* pour *de taille dynamique*

Figure 3.10 – *Représentation graphique d'une boucle* pour *à nombre d'itération fixe ou dynamique.*

Avec la notion de hiérarchie, cette représentation des boucles agissant sur un noyau permet d'itérer sur n'importe quelle sous-application. Néanmoins, construire une sous-application nécessite une description distincte car celle-ci est représentée par un simple noyau dans l'application de niveau supérieur. Or, si la sous-application est relativement simple, on peut désirer ne pas l'extraire dans un noyau séparé et la représenter directement. À cette fin, on autorise leur représentation directement dans une application au sein d'un rectangle arrondi, mettant en évidence la délimitation. Par exemple, une boucle itérant sur deux noyaux parallèles peut être représentée comme sur la figure 3.11.

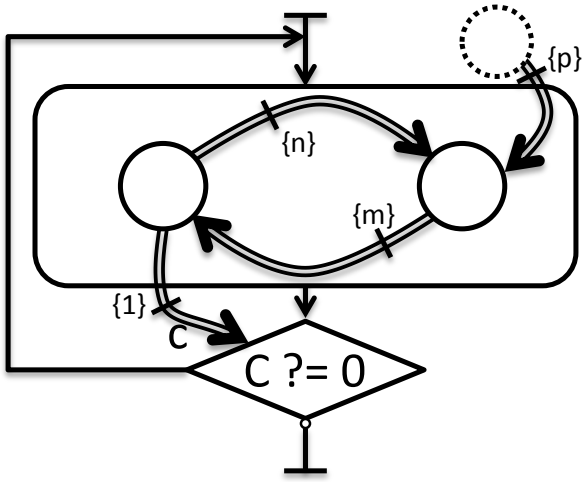


Figure 3.11 – *Représentation graphique d'une boucle* faire ... tant que *opérant sur une sous-application.*

3.1.1.g Environnement

Sur la figure 3.11, on remarque un noyau en pointillés : celui-ci représente l'environnement. Une quantité de données p est donc fournie par l'environnement. Les données échangées avec l'environnement peuvent prendre diverses formes : lecture et écriture de fichiers, accès à des sources de données tels les capteurs, ou encore action sur un périphérique. Enfin, sur cette même figure, on introduit les marqueurs de début et de fin, représentés par des équerres, permettant de signifier un point d'entrée et une fin pour l'application.

3.1.2 Implémentation des noyaux – Traitement des données

Comme vu précédemment, un noyau peut être représenté indépendamment de l'application soit par un noyau final, soit par une autre application. On s'intéresse ici aux noyaux finaux, c'est-à-dire aux feuilles du graphe d'application, qui agissent directement sur les données. Afin d'autoriser une portabilité maximale, nous utilisons une technique de *virtualisation* de noyaux, c'est-à-dire une manière d'interagir avec eux quelle que soit leur implémentation réelle.

3.1.2.a Virtualisation des noyaux

Dans le cas d'une plateforme homogène, où toutes les ressources de calcul sont les mêmes, un noyau final peut être exécuté sur n'importe quelle ressource. Si la plateforme est hétérogène, en revanche, un noyau final ne pourra être exécuté que sur les ressources compatibles. L'hétérogénéité peut se manifester entre ressources du même type, par exemple des processeurs utilisant des jeux d'instructions différents. Celle-ci peut également exister sur la nature des ressources, comme les

HPRCs proposant à la fois des ressources logicielles et matérielles. L'implémentation de chaque noyau final est alors déterminée pour une exécution sur un type de ressource particulier.

Afin de lever cette limitation, il est possible de distribuer plusieurs implémentations d'un noyau final afin d'autoriser son exécution sur les différents types de ressources présents dans la plateforme. Mais une application dont les implémentations des noyaux ciblent des ressources particulières est étroitement liée à une plateforme. Or, notre souhait est d'assurer une portabilité maximale à nos applications.

Afin d'assurer l'indépendance de l'application vis à vis de la plateforme tout autant que la possibilité de réutilisation, il est possible d'introduire une couche de virtualisation à l'échelle du noyau, comme présenté sur la figure 3.12. On décrit alors un noyau comme un traitement dont on connait l'objectif, mais pas l'implémentation réelle. Il suffit alors de distribuer les implémentations à l'exécution en fonction des ressources disponibles.

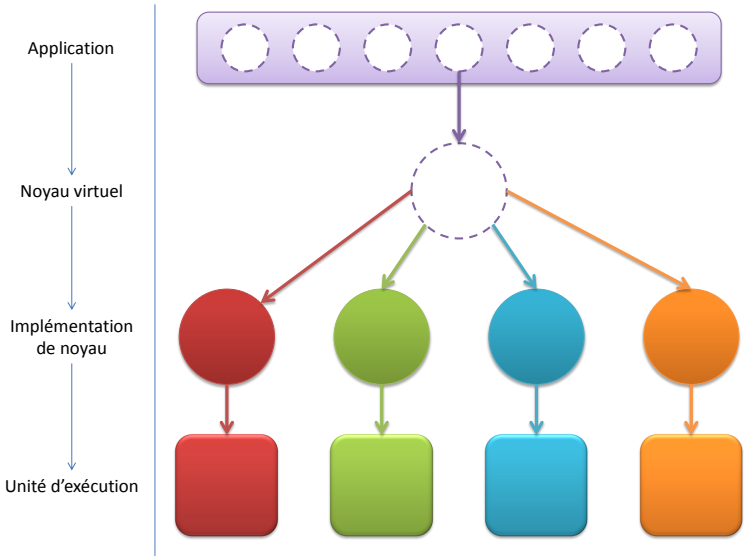


Figure 3.12 – *Introduction d'une couche de virtualisation dans la hiérarchie de l'application.*

Cette couche de virtualisation doit notamment prendre en charge la diversité d'interface des IPs, afin d'assurer la compatibilité entre différentes implémentations. En effet, l'interface de deux IPs existants réalisant le même traitement peut varier fortement, à fortiori si l'un est matériel et l'autre logiciel. Il s'agira donc de gérer ces interfaces de manière transparente au sein de la couche de virtualisation.

L'un des objectifs que l'on se donne est de proposer la réutilisation de blocs de propriété intellectuelle existants lors de la conception d'une application. Il s'agit de permettre, pour un IP existant, de l'utiliser dans une application en tant que noyau, diminuant ainsi le temps de développement, et donc le *time to market*. Ainsi,

le développement d'une application pourra se révéler très rapide si les différentes implémentations des noyaux finaux utilisés existent déjà pour la plateforme visée.

Par ailleurs, proposer plusieurs implémentations permet également de proposer différentes performances. Par exemple, pour des implémentations matérielles, il est possible de disposer de plusieurs niveaux de parallélismes différents pour une même opération. Un choix peut alors être fait à l'exécution en examinant le rapport performance/surface pour déterminer à l'aide d'heuristiques quelle implémentation est la plus adéquate à utiliser dans l'environnement courant.

3.1.2.b Modélisation des noyaux virtuels

Un noyau final peut proposer plusieurs implémentations. En virtualisant celles-ci pour les représenter de manière indépendante de la plateforme, on offre une vue générique ne présentant que ses échanges de données. Ainsi, le noyau définit un certain nombre de *ports* pour représenter les communications et les dépendances de données. Chaque implémentation devra alors définir un descripteur faisant le lien entre les évènements sur un port et l'interaction avec l'interface de l'IP.

Outre les interactions de données, le contrôle agit également sur l'interface de l'IP. Le contrôle permet notamment d'agir sur le *contexte* de l'IP. Dans le cadre de la préemption, le contexte d'une implémentation peut être sauvegardé avant de mettre un noyau en pause. À contrario, la restauration de contexte permet de remettre le noyau dans un état sauvegardé antérieurement, ou bien dans un état initial précis. Afin d'autoriser la migration de tâche, consistant en le transfert d'un noyau d'une ressource vers une autre, il est nécessaire d'avoir une indépendance du contexte vis-à-vis de l'implémentation, tout comme pour les ports.

Une implémentation d'un noyau virtuel est donc constituée des éléments suivants :

- l'IP lui-même,
- le descripteur de port,
- le descripteur de contrôle.

Le descripteur de contrôle d'une implémentation doit ainsi définir les opérations de sauvegarde et de restauration de contexte, ainsi que les opérations sur l'état permettant de démarrer, arrêter, suspendre et reprendre le traitement. Le descripteur de port, quant à lui, doit gérer la lecture et l'écriture sur les ports en fonction des opérations de communication.

3.2 Spécification d'un modèle de plateforme d'exécution

L'un des objectifs consiste à proposer une plus large flexibilité pour une application, notamment à travers l'usage de la reconfiguration dynamique partielle. Or, on

veut pouvoir proposer cette fonctionnalité de manière *transparente* et compatible avec la *réutilisation* d'IPs existants. La transparence consiste à développer l'application sans avoir à se soucier de l'implémentation finale : on construira l'application de la même manière, que les noyaux soient logiciels ou matériels, et quel que soit le jeu d'instruction ou la technologie reconfigurable. La réutilisation permet d'utiliser, pour les implémentations de ces noyaux pour une ressource particulière, des IPs déjà existants.

Le matériel reconfigurable le plus répandu est le FPGA, qui permet de constituer un circuit logique quelconque dans la limite des ressources disponibles. Certains proposent la reconfiguration dynamique partielle [89], qui autorise une modification d'une partie du composant sans interférer avec le fonctionnement des éléments n'étant pas modifiés. Ce composant sera de toute évidence au centre de la plateforme d'exécution. On différenciera la plateforme théorique, décrite dans cette section et devant être générique, des implémentations de cette plateforme. On donne le nom de *Simple Parallel platform for Reconfigurable Environment* (SPoRE) à la plateforme théorique. Les plateformes pratiques seront donc des *implémentations* de SPoRE.

Notre projet met en avant la portabilité entre plateformes et la facilité d'utilisation des PEs existants. À cette fin, n'importe quel type de plateforme permettant l'exécution de noyaux logiciels et/ou matériels reconfigurables doit pouvoir être exploité. Ainsi, il s'agira de définir une plateforme suffisamment générique pour pouvoir être agnostique en termes de matériel utilisé.

3.2.1 Architecture distribuée et hiérarchie

Afin de satisfaire différents besoins en termes de capacité de calcul, et notamment supporter des applications massivement parallèles, SPoRE doit pouvoir supporter une mise à l'échelle. Pour cela, on reprendra la topologie réseau des HPRCs, dans laquelle des nœuds sont liés via un bus. Ceci permet notamment des tailles différentes d'une implémentation à une autre, mais également une gestion dynamique des nœuds au sein d'une implémentation. Chaque nœud est formé d'un ensemble de PEs et de ressources reconfigurables, que nous désignerons sous le nom de *cellules*. En effet, il s'agit de définir les différents types de nœuds existants dans la plateforme ainsi que l'architecture interne de ceux-ci et la structure des cellules.

3.2.2 Architecture globale

Nous considérons qu'une plateforme d'exécution est un ensemble de nœuds reliés par un bus. L'architecture générale de celle-ci est représentée sur la figure 3.13.

Outre les nœuds sur lesquels seront exécutés les calculs, il est nécessaire de disposer d'un nœud de coordination, le *nœud maître*. Celui-ci sera en charge de l'attribution des différentes parties de l'application aux différents nœuds de calcul.

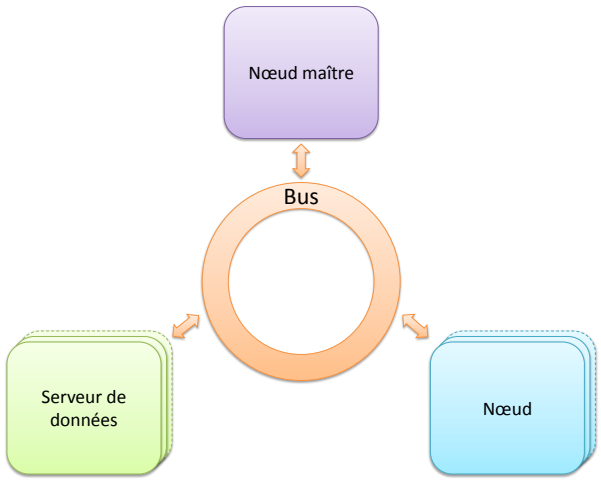


Figure 3.13 – *Vue de la plateforme théorique SPoRE.*

Celui-ci doit disposer d'un système d'ordonnancement à même de séparer une application en différents éléments et de les déployer sur les nœuds. Ce système sera désigné par le nom d'*ordonnanceur global*.

Également, et à moins de distribuer préalablement les différentes parties de l'application, ce qui impliquerait un ordonnancement statique, il faut disposer d'un nœud de stockage sur lequel seront entreposées les données correspondant au contenu de l'application. Les différents nœuds pourront alors s'y connecter afin de récupérer les éléments nécessaires à l'exécution. On parlera alors d'un nœud de type *serveur de données*, sur lequel seront entreposés les différents descripteurs, fichiers de configuration et exécutables composant l'application. En fonction de la topologie de la plateforme, il sera possible de disposer de plusieurs serveurs de données fonctionnant en miroir afin de faciliter un accès simultané, ou d'éviter un lien à bande passante faible. Les serveurs de données devront aussi permettre de centraliser les résultats des applications.

Afin de mettre en œuvre la simplicité d'utilisation, un point d'entrée unique pour l'exécution d'une application doit être proposé à l'utilisateur. Celui-ci sera situé sur le nœud maître : il suffira à l'utilisateur d'indiquer la référence de l'application à exécuter, qui sera alors gérée par l'ordonnanceur global.

3.2.3 Architecture des nœuds

Un nœud est un ensemble de cellules de calcul qui doit être coordonné afin de s'intégrer dans la plateforme. À cette fin, le nœud devra disposer d'une structure de contrôle permettant la connexion avec les autres nœuds ainsi que la gestion locale. Nous introduisons donc une cellule spéciale, la *cellule hôte*, qui sera en charge de la gestion du nœud, et sera son interface avec l'ensemble de la plateforme. Une représentation simple d'un nœud est présentée sur la figure 3.14.

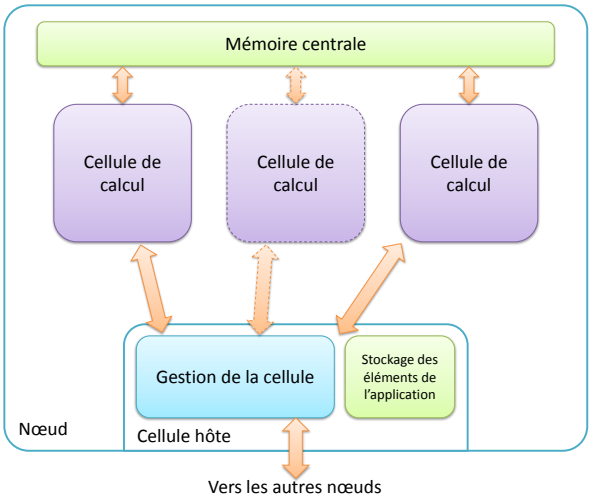


Figure 3.14 – *Architecture d'un nœud SPoRE.*

Ainsi, la cellule hôte supporte la communication entre le nœud et le reste de la plateforme via le bus central. Elle a également une fonction d'ordonnancement local, distribuant les noyaux qui lui sont attribués sur les différentes cellules. À cette fin, elle est également en charge de la reconfiguration des cellules matérielles. La cellule hôte doit être reliée à chacune des cellules de calcul afin de connecter ces dernières à l'ensemble du système.

Par ailleurs, les cellules de calcul doivent disposer d'un accès à la mémoire centrale du nœud afin d'autoriser une communication de type dépendance de données sans transferts. Chaque cellule peut disposer en sus d'une mémoire locale pour un accès plus rapide, la mémoire centrale partageant aussi la bande passante, ce qui limite les accès simultanés par plusieurs cellules. Pour la suite, on utilise le terme de *bus* au sens large, c'est-à-dire un système central d'échange de données. Il est possible d'architecturer les différentes cellules autour d'un bus afin de permettre une communication point à point entre elles si nécessaire, ou bien de ne relier chaque cellule de calcul qu'à la cellule hôte. Ce choix dépendra de l'implémentation, une communication par message préférant un bus commun et une communication de type mémoire partagée ne le nécessitant pas.

3.2.4 Découpage hiérarchique du nœud

Selon l'architecture utilisée pour implémenter un nœud, tout ou partie de celui-ci peut être situé sur du matériel reconfigurable. Dans le cas idéal, où un FPGA est utilisé comme base du nœud, il est possible de représenter celui-ci en utilisant une hiérarchie de couches reconfigurables, de manière à permettre une flexibilité maximale. Encore à l'état théorique, bien que de nombreuses recherches soient menées à ce sujet, la reconfiguration hiérarchique consiste à disposer de zones reconfigurables au sein d'une zone reconfigurable. Un exemple de découpage hiérarchique

d'un nœud implémenté sur un FPGA est présenté sur la figure 3.15.

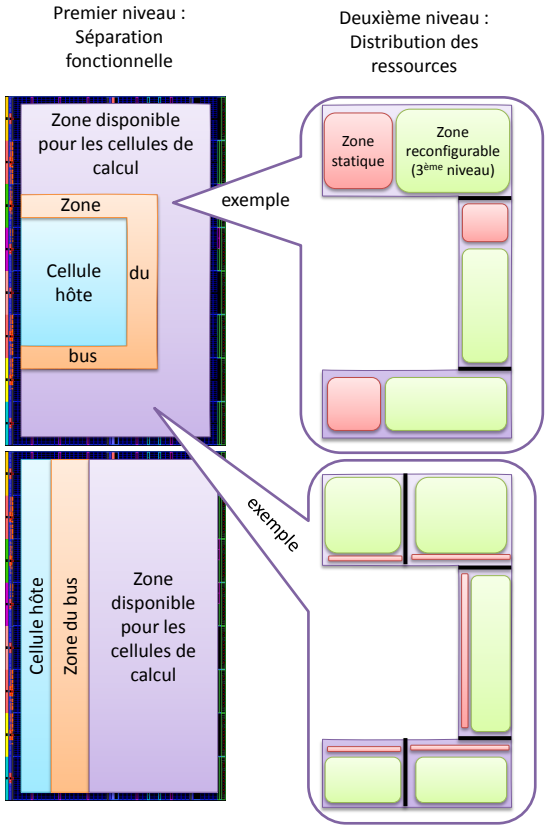


Figure 3.15 – *Exemple de découpage hiérarchique d'un nœud SPoRE.*

Le niveau le plus bas de cette hiérarchie consiste en une séparation de la cellule hôte, des bus et des cellules de calcul. Ce premier niveau permet de séparer ces trois éléments de base du nœud afin de réserver l'espace disponible pour chacun de manière flexible en fonction des besoins. Il s'agit finalement de distribuer les ressources qui seront réservées à chacun de ces trois éléments. Ce découpage sera généralement statique, du moins en ce qui concerne la cellule hôte. En effet, celle-ci gérant la reconfiguration, elle ne peut se reconfigurer elle-même.

Au dessus de ce premier découpage en zones selon leur fonctionnalité, il est possible de faire plusieurs ajustements. Ainsi, sur les ressources disponibles pour placer les cellules de calcul, il est possible de disposer un nombre variable de cellules, en allouant plus ou moins de ressources à chacune. Pour la zone dédiée au bus, il est possible de proposer plusieurs topologies, notamment en fonction du nombre de cellules de calcul, par exemple en instanciant un NoC lorsque le nombre de cellules est important.

Enfin, le troisième et plus haut niveau concerne le contenu des cellules elles-mêmes, qui sont bien entendu reconfigurables afin de permettre d'implémenter des PEs logiciels ou matériels. La cellule dispose d'une zone statique, définie au

niveau 2 et permettant le lien entre le thread et l'ensemble du nœud, et d'une zone dynamique, formant le niveau 3 et accueillant un noyau.

3.2.5 Gestion de la mémoire et des communications

Selon le type de noyaux utilisés dans une application, différents types de communication et d'utilisation de la mémoire peuvent apparaître. Au sein des applications parallèles, les deux types de communications les plus utilisés sont MPI et OpenMP.

À l'échelle d'un nœud, OpenMP permet une communication par mémoire partagée très simple du point de vue développeur car automatiquement prise en charge par le compilateur. En revanche, cette forme de communication n'est pas aisément généralisable aux noyaux matériels. En effet, la puissance d'OpenMP est d'offrir une communication implicite dont le développeur n'a pas à s'inquiéter, ce qui n'est pas applicable au matériel, à moins d'utiliser des librairies spécifiques et relativement peu portables. Par ailleurs, l'un des aspects de la performance d'OpenMP est d'utiliser pour la communication la migration de pages mémoires, ce qui nécessite l'utilisation de mémoire virtuelle. Or, si la gestion de la mémoire virtuelle est intégrée dans les systèmes d'exploitation logiciels, la communication entre logiciel et matériel utilisant la mémoire virtuelle reste très ardue. Nous réserverons donc la communication par mémoire partagée aux cellules logicielles s'exécutant en SMP.

À l'inverse, MPI, de par sa nature de communication directe entre noyaux permet un support matériel relativement aisé, voire l'utilisation de bus supportant MPI directement de manière matérielle. Le support de MPI par SPoRE semble donc réalisable, et qui plus est souhaitable en raison de son statut de standard *de facto*.

Les communications de type dépendance de données, quant à elles, devront être gérées par l'environnement de contrôle, c'est-à-dire la cellule hôte. Celle-ci devra également coordonner l'utilisation de la mémoire centrale en attribuant à chaque cellule une portion en fonction de ses besoins. Une cellule peut également posséder une mémoire locale pour ses résultats intermédiaires, mais les résultats finaux devront être stockés dans cette mémoire centrale.

3.2.6 Positionnement

Dans cette section, nous allons étudier le positionnement de la plateforme SPoRE par rapport aux différentes plateformes existantes présentées précédemment.

La plateforme proposée par Æther [71] est entièrement modulaire, disposant de plusieurs couches architecturales reconfigurables. Il est ainsi possible de modifier les ressources de base (cells) aussi bien que leurs relations (composants et SANEs). La plateforme SPoRE, quant à elle, ne reprend pas ce principe de modularité multi-niveaux, lui préférant la notion de cellule en tant qu'ensemble de ressources sta-

tiques ou reconfigurables. Dans certains cas néanmoins, il est possible de faire le parallèle entre Æther et SPoRE. En effet, dans la plateforme Æther, la granularité des cells peut varier jusqu'à être formée d'un processeur, ce qui correspond à une cellule SPoRE statique. Par ailleurs, si la granularité des cells tombe à la taille d'un CLB, on peut alors envisager le composant Æther, qui rassemble plusieurs cells comme équivalent à une cellule SPoRE reconfigurable. Dans ce cas, le SANE Æther peut être rapproché de la notion de nœud dans la plateforme SPoRE. Les deux plateformes présentent donc des points communs, leur principale distinction s'opérant sur la hiérarchie. Néanmoins, Æther reste une plateforme théorique, utilisée en simulation, là où SPoRE a été réellement implémentée. Ainsi, Æther dispose d'une hiérarchie totalement modulable quand SPoRE préfère réserver deux niveaux principaux, nœud et cellule, disposant chacun d'éléments de supervision destinés à gérer la couche de contrôle de l'application. C'est ainsi que la cellule hôte présente dans les nœuds SPoRE offre un niveau de gestion local qui n'est pas présent dans la plateforme du projet Æther.

La plateforme BORPH [79] permet d'intégrer des éléments matériels reconfigurables dans un système logiciel UNIX. Cette possibilité est un grand atout de la plateforme : en effet, le système de processus UNIX est un standard très utilisé. Cette intégration d'éléments matériels prend la forme de processus en tout point identiques aux processus logiciels, et donc permettant l'interaction entre eux de manière native. Si les avantages de cette technique sont certains en terme d'intégration, la difficulté ici concerne la réutilisation d'IPs matériels déjà existant. En effet, l'un des piliers de la communication dans les processus UNIX est la transmission de données via les flux standards. Or, la notion de flux n'est pas forcément facilement adaptable à l'interface d'un IP matériel. Si celle-ci sera aisément applicable aux IPs réalisant des opérations sur des flots de données en mémoire, elle peut néanmoins difficilement être adaptée aux PEs ne fonctionnant que sur la base d'opérations de type registre. Ainsi, l'utilisation de tels IPs dans ce système pourra être difficile. La plateforme SPoRE, quant à elle, propose une réutilisation facilitée d'IPs déjà existants. Cette plateforme est donc très intéressante de par ses aspects d'intégration dans un système standardisé, mais ce même point fort peut également se révéler être une contrainte. En revanche, lors de la création d'un nouvel IP, une conception dédiée à la plateforme permettra de tirer tous les avantages de la notion de processus matériel.

Egret [7] est une plateforme modulaire, ce qui signifie qu'il est possible de lui ajouter des fonctionnalités en y branchant des cartes d'extension. Celle-ci dispose de capacités reconfigurables par l'utilisation de FPGAs. La plateforme fonctionne en utilisant un Linux embarqué, qui contient un driver pour l'ICAP. Une reconfiguration peut ainsi être réalisée par un simple transfert du fichier de configuration vers le driver de l'ICAP. Cette plateforme supporte les applications temps-réel strict, via l'utilisation de RTLinux, ce qui permet une adaptation aux utilisations

embarquées. En revanche, le support de la reconfiguration partielle n'est pas mentionné. En effet, les modules présents sont identifiés au démarrage, et l'architecture chargée sur le FPGA est réalisée en fonction des modules présents. Le système est donc plug-and-play à froid, c'est à dire que l'identification des périphériques est automatique, mais réalisée uniquement au démarrage du système. Ceci ne permet donc pas un usage auto-adaptatif, qui nécessite que les IPs présents sur le FPGA puissent évoluer au cours de l'exécution. Avec la plateforme SPoRE, nous proposons le support de la reconfiguration dynamique partielle, ainsi que des éléments natifs pour le support de l'auto-adaptativité.

Les auteurs de ReMAP [84] proposent une plateforme logicielle épaulée par un réseau matériel reconfigurable permettant la communication ainsi que des fonctionnalités d'accélération matérielle. Cette solution a pour principal mérite de disposer d'un réseau dynamique mettant en lien les PEs en fonction des besoins de l'application. L'utilisation de ce NoC reconfigurable est très intéressante pour implémenter des fonctionnalités de communication de type réduction, par exemple dans MPI. En revanche, à l'exception de la communication, l'approche générale de cette plateforme reste centrée sur le logiciel, le matériel n'étant vu que comme une capacité d'appoint pour accélérer certaines tâches. Avec la plateforme SPoRE, nous remettons en cause la prédominance du logiciel en ne prenant pas en compte la nature des unités d'exécution lors de la conception de l'application. Dans une tendance à l'augmentation de l'hétérogénéité des systèmes, l'approche centrée sur le logiciel est-elle toujours d'actualité ? On peut également faire la même remarque sur les plateformes de type MPSoCs.

OS4RS [53] s'emploie à mettre toutes les ressources disponibles dans une plateforme à disposition des applications. On peut ainsi utiliser tant des processeurs que des DSPs ou des FPGAs. Cette unification des PEs dans un ensemble de ressources mises à disposition des applications rejoint le principe de SPoRE consistant à repenser les relations entre les PEs en fonction de leur nature. Ainsi, une application peut utiliser uniquement des ressources matérielles si elle le souhaite, brisant la domination du logiciel généralement constatée dans les plateformes actuelles. Par ailleurs, l'utilisation de la reconfiguration dynamique partielle permet une gestion en ligne des IPs matériels. Par rapport à cet OS et à sa gestion d'un système hétérogène, SPoRE apporte la distribution sur plusieurs nœuds liés par un réseau. La structure hiérarchique de SPoRE permet ainsi de découper une application afin de permettre l'exécution des différents noyaux sur des nœuds séparés.

3.3 Conclusion

Dans ce chapitre, nous avons défini un modèle d'application adapté à notre objectif. Celui-ci permet d'exprimer le parallélisme en utilisant un graphe de noyaux indépendamment de l'implémentation de ceux-ci, en séparant la partie contrôle de

la partie traitement. Cette indépendance permet la portabilité de l'application dans son ensemble d'une plateforme à une autre, à condition que les noyaux disposent d'implémentations pour les ressources de cette plateforme. À cette fin, nous avons souhaité pouvoir permettre la réutilisation d'IPs existants, dans le but de minimiser ce travail de portabilité en utilisant des implémentations déjà existantes en tant que noyaux.

Par la suite, nous avons présenté le type de plateforme à supporter, en indiquant une architecture permettant d'organiser les ressources. Cette architecture rejoint celle des HPCs, HPRCs et grilles de calcul sous la forme de nœuds rassemblés en réseau, chaque nœud disposant d'un certain nombre de ressources de calcul, ou cellules. Ces cellules peuvent être un PE existant ou une zone reconfigurable sur laquelle peut être instancié un PE. Cet ensemble de spécifications forme la plateforme théorique SPoRE, dont nous allons maintenant nous atteler à détailler deux implémentations.

Chapitre 4

Première implémentation : plateforme logicielle MPI

Sommaire

A la suite de la définition des spécifications de SPoRE, il convient d'implémenter celle-ci. Dans un premier temps, l'architecture globale de la plateforme SPoRE doit être validée. En effet, la répartition en nœuds et en cellules, adaptée aux HPCs, doit prouver sa faisabilité sur une plateforme de type FPGA. Afin de ne pas mélanger les problématiques, nous avons fait le choix de concevoir cette première implémentation sur la base de PEs uniquement logiciels, ceci afin d'éviter d'introduire la reconfiguration dynamique partielle.

En effet, en utilisant des PEs logiciels, il n'est pas nécessaire de recourir à la reconfiguration matérielle pour en modifier le comportement. La reconfiguration logicielle, consistant à remplacer le code exécutable déployé sur le PE, nous permettra de modifier le comportement des PEs sans avoir recours à la RDP. La reconfiguration matérielle sera introduite, dans la deuxième implémentation qui sera présentée dans le prochain chapitre.

Dans cette plateforme, nous utilisons MPI pour la communication entre cellules. En effet, MPI étant une spécification très répandue, de nombreuses applications l'utilisent, et seront donc compatibles avec cette implémentation de SPoRE. L'idée est de pouvoir tester la structure générale de SPoRE en utilisant des applications existantes, et de mettre en lumière les avantages et inconvénients.

De par son architecture logicielle/MPI, la structure de cette implémentation s'apparente à celle des HPCs. Nous avons donc choisi de nommer cette première implémentation de SPoRE *Software HPC Platform* (SHP). Dans un premier temps, nous présentons les caractéristiques, tant matérielles que logicielles, de l'implémentation SHP en section 4.1, pour ensuite nous intéresser à la validation et l'évaluation de la plateforme à l'aide d'une application de test en section 4.2 et finalement en tirer les conclusions en section 4.3.

4.1 Caractéristiques de l'implémentation

Nous allons tout d'abord détailler les choix d'implémentation réalisés pour SHP. Comme pour toute conception, une part des éléments provient de réutilisation d'IPs existants, tandis que d'autres ont été conçus spécialement pour cette implémentation. Pour cette implémentation, la partie matérielle sera issue d'éléments existants tandis que la partie logicielle inclura des éléments nouveaux. Dans cette section, nous commençons par présenter l'architecture matérielle utilisée pour concevoir cette plateforme et la manière dont elle est agencée, puis nous nous intéresserons à la couche logicielle.

4.1.1 Architecture matérielle

Nous avons choisi de baser notre plateforme sur des cartes de développement Xilinx ml507. En effet, celles-ci disposent de tous les éléments nécessaires pour

construire les nœuds de notre plateforme. Les cartes ml507 proposent un environnement de développement et de prototypage articulé autour d'un FPGA Virtex 5 de type fx70t. Le Virtex 5 fx70t propose 5 600 CLBs contenant chacun 8 LUTs à 6 entrées et 8 bascules, ainsi que 5 328 Kio de blocs RAM (BRAM), correspondant à de la mémoire intégrée au sein du composant. Ce composant intègre également un processeur PowerPC 440 en statique. Autour du FPGA, la carte ml507 contient la connectique nécessaire à un environnement embarqué, et notamment des ports Ethernet, série et PCI Express. La carte met également à disposition de la mémoire vive, à raison de 256 Mio de DDR2.

En instanciant sur le FPGA les éléments logiques nécessaires à notre système, nous disposons d'une base pour construire un nœud SPoRE. Ces nœuds pourront s'interfacer par la suite via un bus de connexion, par exemple Ethernet ou PCI Express, selon les besoins, pour former la plateforme complète.

4.1.1.a Architecture de la cellule hôte

Selon notre spécification, l'architecture d'un nœud est constituée de cellules de calcul articulées autour d'une cellule hôte.

La cellule hôte est destinée à coordonner et ordonnancer le travail des cellules de calcul, ainsi qu'à permettre la communication entre les différents nœuds. À ce titre, nous avons choisi d'implémenter celle-ci de manière logicielle en utilisant le PowerPC comme base pour héberger une implémentation MPI. La cellule hôte est donc articulée autour du PowerPC, qui fonctionne à la fréquence de 400 MHz. Afin que celle-ci soit fonctionnelle, il est nécessaire de lui adjoindre des éléments matériels, placés sur un bus système PLB comme indiqué sur la figure 4.1.

Tout d'abord, il est nécessaire, tant pour la cellule hôte que pour les cellules de calcul, de disposer d'un accès à la mémoire partagée. Pour cela, on utilise un contrôleur mémoire à plusieurs accès (Multi-Port Memory Controller – MPMC), qui dispose d'un arbitre permettant de partager l'accès à la mémoire vive. Afin que le système dispose d'un moyen de stockage pérenne, il est nécessaire d'ajouter une mémoire de masse. Celle-ci sera implémentée à l'aide du lecteur de cartes CompactFlash intégré à la carte, en disposant un contrôleur SysACE connecté sur le bus de la cellule hôte.

La cellule hôte doit également disposer d'un moyen de communication pour pouvoir accéder à ses pairs, ce qui sera réalisé via un contrôleur Ethernet. Enfin, en sus de l'accès Ethernet, nous intégrons un moyen de communication direct avec la carte, qui permettra la communication et le debug même en cas de problèmes d'accès via le réseau. Pour cela, on dispose d'un UART qui permettra d'accéder à l'OS en mode console via le port série. Finalement, on adjoint à ces éléments un gestionnaire d'interruption qui permettra de coordonner les évènements venant des différents ports (Série, Ethernet, ...).

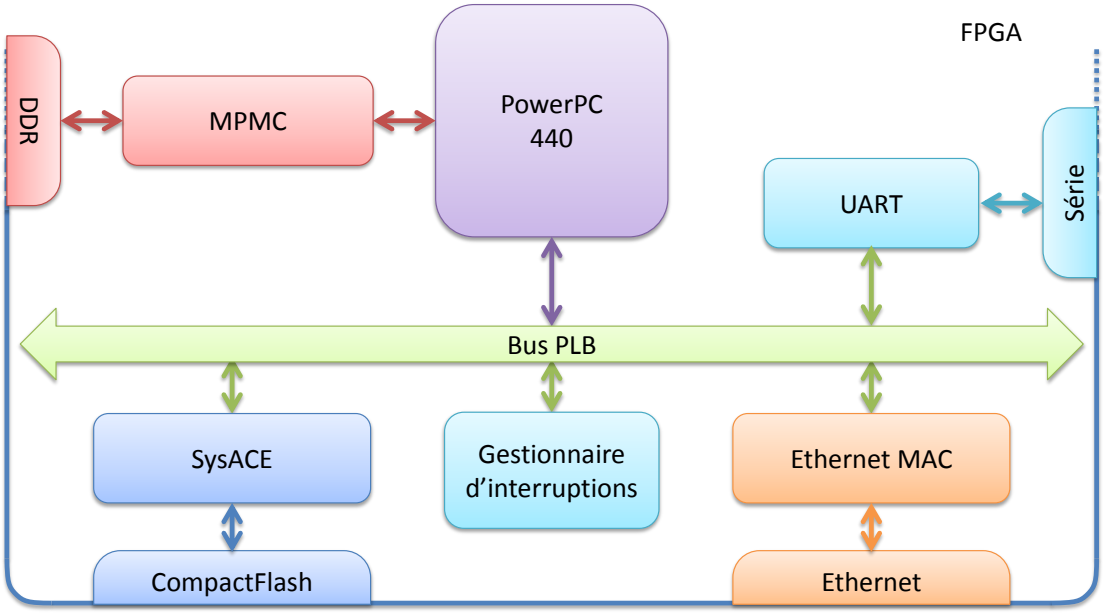


Figure 4.1 – *Éléments constitutifs de la cellule hôte.*

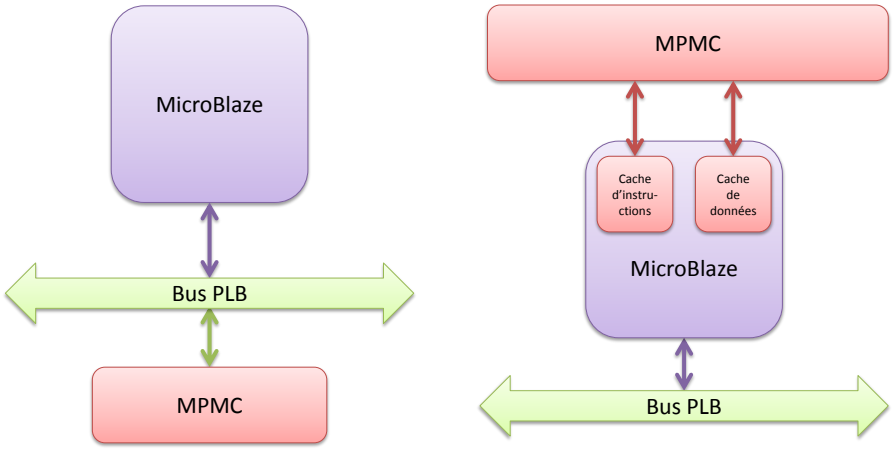
4.1.1.b Architecture des cellules de calcul

Pour l'architecture spécifique de l'implémentation SHP, l'implémentation des cellules de calculs est faite à base de PEs logiciels. Pour cela, nous avons choisi le processeur reconfigurable *MicroBlaze* [90] de Xilinx. Ce processeur « softcore » peut-être instancié à la demande sur des ressources reconfigurables, et dispose de nombreuses options pour le paramétrer selon nos besoins. Il est par exemple possible d'intégrer une unité de calcul en virgule flottante (Floating Point Unit – FPU), des circuits de multiplication et de division, de dimensionner la taille du cache, etc.

Bien que ce processeur ne soit pas prévu pour réaliser du calcul intensif, mais plutôt des opérations de contrôle, sa nature et sa flexibilité en font un bon choix pour une plateforme de test. En effet, nous ne cherchons pas ici à obtenir de fortes performances, mais simplement à valider l'architecture. Il sera toujours possible par la suite de remplacer ces processeurs par des PEs matériels, ou bien par d'autres PEs logiciels softcores plus adaptés au HPC.

Nous choisissons une configuration des MicroBlazes complète. On configure donc ceux-ci avec une FPU étendue, un multiplicateur sur 64 bits et un diviseur, et on fait fonctionner celui-ci à une vitesse de 100 MHz. De plus, nous choisissons une taille de cache de 8 Kio.

Le MPMC disposant au maximum de 8 accès, et un étant réservé pour la cellule hôte, il reste donc 7 branchements disponibles. Selon le choix de l'activation du cache ou non sur les MicroBlazes, ceux-ci utilisent un ou deux accès mémoire, comme indiqué sur la figure 4.2. En effet, si le cache n'est pas activé, l'accès à la



(a) *Cellule de calcul sans cache* **(b)** *Cellule de calcul avec cache*

Figure 4.2 – *Architecture des cellules de calcul avec et sans cache.*

mémoire se fait via un bus PLB intermédiaire, de la même manière que le PowerPC, et utilise donc un seul accès du MPMC. En revanche, comme le cache utilise une architecture de Harvard, l'activation du cache nécessite l'utilisation de deux bus point-à-point, un pour chaque type de cache, instructions et données, ce qui monopolise alors deux accès MPMC. Selon l'activation du cache ou non, on dispose alors d'une capacité d'accueil en cellules de calcul de 3 si le cache est activé ou de 7 avec cache désactivé.

Enfin, on fournit au MicroBlaze une petite mémoire locale, instanciée à l'aide de composants de type BRAM. Celle-ci sera utilisée pour stocker le programme local résidant sur la cellule et permettant son fonctionnement. Plus d'informations seront données sur le programme principal des cellules de calcul dans la section 4.1.2, qui concerne l'architecture logicielle.

4.1.1.c Communication intra-nœud

Il est également nécessaire de prévoir un moyen de communication entre les cellules, qu'elles soient situées sur le même nœud ou non. Pour communiquer avec une cellule d'un autre nœud (communication *distante*), il faudra transiter par la cellule hôte, qui initiera alors une communication via Ethernet. Une communication inter-nœud se traduit donc par une communication entre cellule de calcul et cellule hôte au niveau local. En revanche, pour une communication avec une cellule du même nœud (communication *locale*), deux possibilités existent : une communication directe entre cellules, ou une communication via la cellule hôte.

S'agissant de systèmes à mémoire partagée, l'essentiel de la communication au sein des nœuds peut être faite via celle-ci. Néanmoins, il n'est pas possible ici d'intégrer directement une méthode de communication de type OpenMP. En effet,

celle-ci nécessite un système d'exploitation commun, alors que nos cellules sont indépendantes. Intégrer OpenMP demanderait donc un très gros effort d'adaptation et de réécriture des compilateurs qui n'est pas compatible avec la vocation de simple test de cette plateforme. Nous utiliserons donc également MPI pour la communication locale.

De ce fait, la communication directe entre cellules se résume à des opérations de contrôle, principalement pour signifier la disponibilité d'un message MPI. La quantité de donnée échangée directement est donc minime, la plus grande quantité transitant par la mémoire. Il n'est donc pas nécessaire d'ajouter un lien direct entre les cellules de calcul, l'échange pouvant être réalisé via la cellule hôte, avec laquelle le lien est forcément présent.

Dans ces conditions, on obtient donc un graphe en étoile, avec la cellule hôte au centre, reliée à chaque cellule de calcul. Il est ensuite nécessaire de dimensionner ce lien. Pour les communications distantes, il est possible de réaliser les échanges entre la cellule hôte et les cellules de calcul de la même manière que les échanges entre cellules de calcul, c'est-à-dire en transitant par la mémoire vive.

En effet, lorsqu'une cellule de calcul doit envoyer des données, celles-ci sont stockées en mémoire. Pour les envoyer, il suffit de signaler à la cellule hôte l'adresse où elles sont stockées pour que celle-ci puisse initier la communication. Les échanges entre cellule hôte et cellules de calcul se résument donc eux aussi à des opérations de contrôle, réduits en termes de quantité de données. Le lien entre ces cellules n'a donc pas besoin d'être de grande capacité.

Nous avons choisi d'implémenter ce lien sous la forme de files (First In, First Out – FIFO) reliant le bus de la cellule hôte à ceux des cellules de calcul : les MailBoxes. Une MailBox contient deux FIFOs reliant les bus, chacune dans un sens. Celles-ci permettent d'échanger des mots de 32 bits entre les cellules, comme présenté sur la figure 4.3.

4.1.1.d Occupation des ressources matérielles

Les taux d'occupations des ressources principales pour chaque cellule sont indiqués dans la table 4.1. La quantité totale de ressources utilisées par un nœud est

Resource	Cellule de calcul sans cache	Cellule de calcul avec cache	Cellule hôte
LUTs	3 870 (8,6%)	4 441 (9,9%)	2 559 (5,7%)
BRAMs	17 (11,5%)	34 (23,0%)	23 (15,5%)
DSP48Es	6 (4,7%)	6 (4,7%)	0 (0,0%)

Tableau 4.1 – *Ressources utilisées par les cellules de la plateforme SHP. Les pourcentages indiquent l'occupation du composant Virtex 5 fx70t.*

présentée dans la table 4.2. On constate que la ressource limitante est la BRAM.

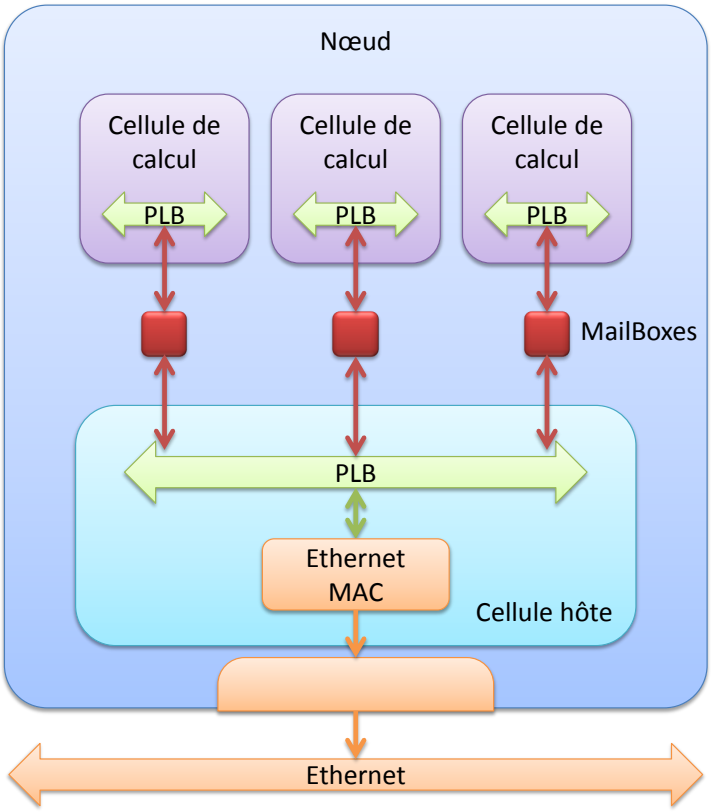


Figure 4.3 – *Communication entre les nœuds.*

En effet, même si le MPMC avait plus que 8 ports, il serait impossible de placer plus de cellules, en raison de cette limitation.

4.1.1.e Bilan matériel

Ainsi, l'architecture matérielle de la première implémentation de SPoRE est entièrement composée d'éléments existants. Cette partie matérielle n'a pas requis la création de nouveaux composants ; nous nous sommes contentés d'assembler des IPs existants. Il s'agit donc principalement d'un travail de réutilisation, afin d'assembler des composants selon une nouvelle forme.

Ceci nous a permis de concevoir la partie matérielle de la plateforme SHP rapidement à l'aide des outils de conception fournis par Xilinx pour le développement sur ses dispositifs. L'essentiel du travail a été de choisir les composants à utiliser pour se conformer au cahier des charges de SPoRE réduit à une utilisation d'application logicielles MPI. Sur cette base, nous avons pu nous concentrer sur la partie logicielle, en charge de l'exécution des noyaux.

Resource	Nœud sans cache	Nœud avec cache
LUTs	29 649 (66,2%)	15 882 (35,5%)
BRAMs	142 (95,9%)	125 (84,5%)
DSP48Es	42 (32,8%)	18 (14,1%)

Tableau 4.2 – *Ressources utilisées par un nœud de la plateforme SHP. Les pourcentages indiquent l'occupation du composant Virtex 5 fx70t.*

4.1.2 Architecture logicielle

Sur la base de cette architecture matérielle, il est maintenant nécessaire de définir les composants logiciels qui permettront l'exploitation de cette plateforme. Il faudra pour cela construire l'environnement de la cellule hôte et des cellules de calcul, ainsi que la gestion de la communication entre elles.

4.1.2.a Éléments existants

Concernant le système d'exploitation déployé sur la cellule hôte, nous avons choisi un Linux embarqué. En effet, celui-ci présente plusieurs avantages : il est notamment disponible gratuitement et jouit d'une large communauté pour le support. De plus, de nombreuses implémentations de MPI sont disponibles pour ce support, permettant une intégration directe du mécanisme d'ordonnancement et de communication.

Pour le noyau Linux, Xilinx met à disposition un noyau intégrant les pilotes pour les principaux périphériques de la firme [59]. Ainsi, le support du module Ethernet ou encore du SysACE est immédiat en utilisant cette version. La version 2.6.34 du noyau Linux a été retenue en raison de sa stabilité. En effet, les versions ultérieures disponibles à cette date posaient différents problèmes : impossibilité de démarrer, incompatibilité avec le système de fichiers, etc.

Pour le système de fichiers, nous avons retenu Buildroot [77]. Il s'agit d'un utilitaire permettant de sélectionner les différents paquets, exécutables et bibliothèques dont l'on souhaite disposer. Celui-ci offre notamment la plupart des utilitaires UNIX de base via l'intégration de BusyBox [74], ainsi qu'un support des bibliothèques C avec μClibc [76].

Concernant l'implémentation de MPI, notre choix s'est porté sur MPICH du fait de son support bien développé et de sa gratuité. Ceci nous a d'ailleurs été utile pour corriger certains bugs liés à l'implémentation très particulière de notre environnement, sur PowerPC 440 embarqué. La communauté nous a ainsi fourni certains correctifs qui nous ont permis de mener à bien notre expérimentation.

4.1.2.b Éléments supplémentaires développés

Sur cette base Linux/MPI, nous avons une première couche permettant de faire fonctionner une application de manière distribuée sur les nœuds. En déployant un ordonnanceur MPI sur un poste de travail relié au réseau formé par les cartes, nous sommes alors capables, à ce stade, de faire fonctionner une application MPI sur les PowerPC.

L'étape suivante consiste donc à distribuer, au niveau de chaque nœud, la charge de travail entre les différentes cellules de calcul. Pour cela, l'application est séparée en deux niveaux, le contrôle, et le traitement. Le contrôle, exécuté sur le PowerPC, concerne le lancement de processus MPI. Le traitement, quant à lui, concerne les noyaux déployés sur MicroBlaze.

C'est ainsi que nous avons développé une couche de compatibilité permettant de rediriger les lancements de processus sur le PowerPC vers des exécutions sur les MicroBlazes. Nous allons nous intéresser dans un premier temps à l'environnement d'exécution déployé sur les cellules de calcul, puis à la communication entre l'hôte et les noyaux, et enfin à la procédure de délégation des noyaux aux cellules de calcul.

Environnement d'exécution des cellules de calcul

En premier lieu, les cellules de calcul doivent disposer d'un environnement d'exécution (*runtime environment*, par la suite appelé runtime). Ce runtime doit notamment être capable de prendre en charge les communications avec la cellule hôte via une MailBox : ordre d'exécution, support des communications MPI, transfert des résultats... Celui-ci sera stocké dans la mémoire locale de la cellule pour un accès direct et indépendant du reste du nœud.

Conçu comme un système de supervision minimaliste, le runtime est lancé au démarrage de la carte, puis attend les ordres d'exécution fournis par la cellule hôte. Un exemple de communication via les MailBoxes est fourni sur la figure 4.4. Lors de la réception d'un ordre d'exécution, le runtime de la cellule de calcul saute à l'adresse mémoire fournie par la cellule hôte, qui contient le noyau à exécuter. Les ordres de communications avec les autres cellules, locales ou distantes, sont transmis via les MailBoxes alors que les données transitent par la mémoire partagée. Enfin, à la fin de l'exécution du noyau, le runtime fournit un signal de fin à la cellule hôte, puis se place à nouveau en attente.

Interface de communication entre cellule hôte et cellules de calcul

Afin de permettre à la cellule hôte de déléguer l'exécution des noyaux aux cellules de calcul, nous avons mis en place une interface à plusieurs couches. Celle-ci permet de faire le lien entre la partie contrôle d'une application, sur la cellule hôte,

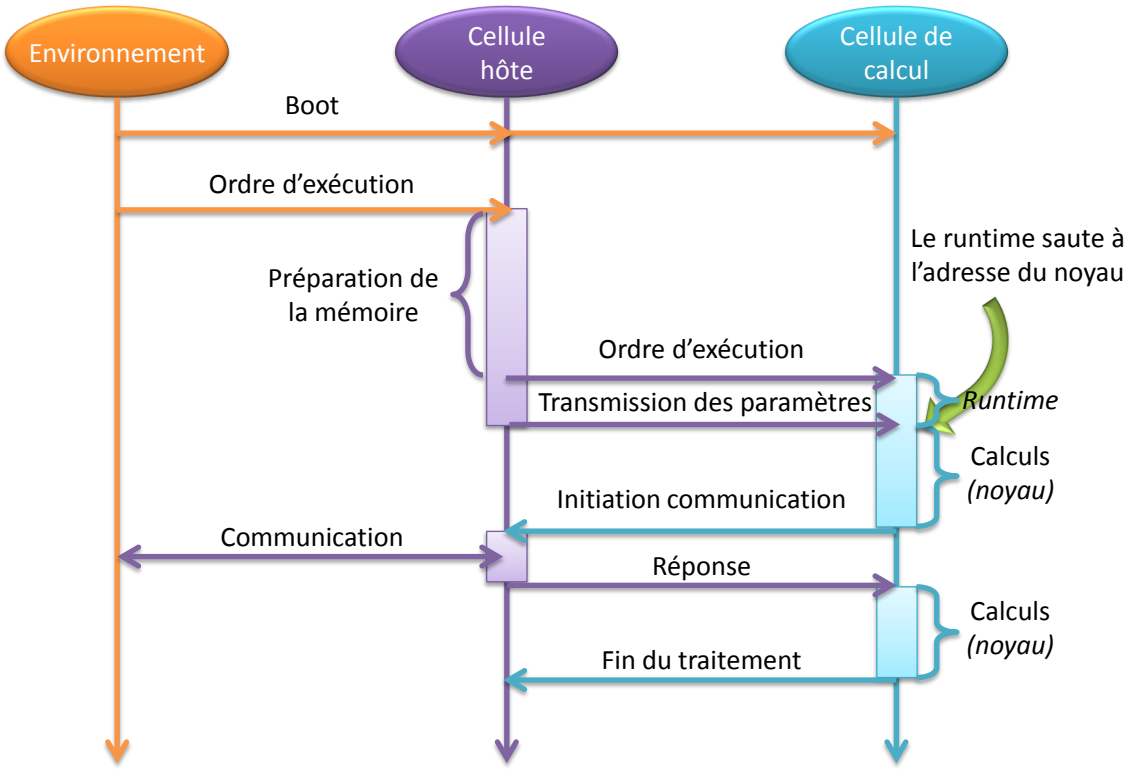


Figure 4.4 – *Étapes de la communication entre la cellule hôte et une cellule de calcul.*

et les noyaux exécutés sur les cellules de calcul. Les différentes couches de l'interface sont le *proxy*, l'*adaptation* et le *pilote*, présentées sur la figure 4.5. Ces couches sont présentes à la fois sur la cellule hôte et les cellules de calcul et permettent la traduction d'appels de haut niveau en opérations de communication via les MailBoxes.

Au plus bas niveau, le pilote permet la gestion des MailBoxes. Coté cellule hôte, le pilote est représenté par un pseudo-fichier linux sur lequel il est possible de réaliser des opérations de lecture et d'écriture, opérations répercutées sur l'IP en écrivant ou en lisant à l'adresse correcte sur le bus. Sur les cellules de calcul, le pilote se résume à des routines opérant des lectures/écritures directement sur le bus. Le pilote contrôle également la validité des opérations en vérifiant la présence de messages et le niveau de remplissage de la FIFO. Enfin, il effectue également les opérations de conversion de l'ordre des octets dans les mots (« endianness ») pour la compatibilité entre processeurs.

Au dessus du pilote, l'adaptation offre un ensemble de routines permettant de transmettre des types de données standard : caractères, entiers, nombres à virgule flottante simple et double précision, chaines de caractères, etc. Cette couche transforme les appels en opérations d'écriture et de lecture sur le pilote correspondant à la taille de la donnée. Par exemple, l'envoi d'un nombre double précision, stocké

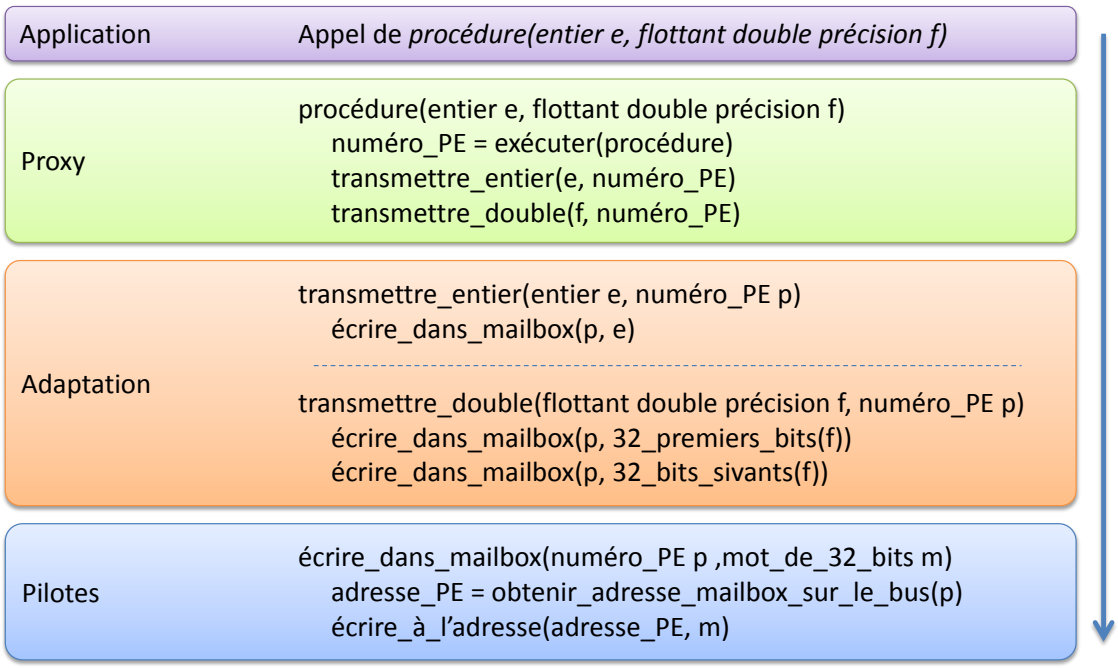


Figure 4.5 – *Les différentes couches de l'interface de communication entre cellule hôte et cellules de calcul.*

sur 64 bits, sera séparé en deux écritures de 32 bits dans la MailBox. Cette couche permet également la transmission des chaînes de caractères, utiles pour afficher des messages de la cellule de calcul sur la console de la cellule hôte durant la phase de debug. Pour ce cas particulier, la transmission du message est précédée par l'écriture de sa taille, afin de permettre au destinataire de savoir combien de lectures il doit opérer.

Enfin, le proxy est le plus haut niveau de l'interface. Fortement lié à l'application contrairement aux deux autres couches, c'est lui qui permet de déporter l'exécution d'un noyau vers une cellule de calcul. Le proxy est constitué de fonctions reprenant la signature des fonctions noyaux, mais en remplaçant leur contenu par un ordre d'exécution sur une cellule de calcul, suivi du transfert des paramètres nécessaires. Par exemple, sur la figure 4.5, on utilise une fonction *procédure*, disposant de deux paramètres, déportée sur une cellule de calcul. L'appel de cette fonction par l'application est récupéré par le proxy, qui lance l'exécution du noyau sur une cellule de calcul. Ensuite, les deux paramètres sont transmis à la couche d'adaptation via les procédures correspondant au type de paramètre.

Dans l'autre sens, les appels à des procédures MPI sur une cellule de calcul sont remplacés par des appels à un proxy MPI qui transmet les demandes à la cellule hôte. Pour cela, on définit un sous-ensemble de routines MPI autorisées sur notre architecture. Cet ensemble contient les routines d'envoi et de réception de messages les plus communément utilisées, ainsi que certaines opérations de réduction.

Procédure d'exécution des processus sur les cellules de calcul

Le code exécuté sur les MicroBlazes est reconfigurable afin de permettre d'exécuter différents noyaux. Pour cela, la cellule hôte charge le code du noyau en mémoire au cours de l'appel à la procédure *exécuter* présenté sur la figure 4.5.

Le code d'un exécutable pour MicroBlaze, de type *Executable and Linkable Format* (ELF) est composé de plusieurs sections : .bss, .text, .rodata, etc., référencées dans l'en-tête. Chaque section doit être placée à une adresse virtuelle particulière.

Lors d'une exécution, l'interface lit l'en-tête du fichier au format ELF pour en identifier les sections et extraire leurs adresses. Le contenu exécutable en est alors copié en mémoire vive à une adresse dépendant de celle de la section et d'une constante propre à chaque cellule, comme indiqué sur la figure 4.6. Enfin, l'interface

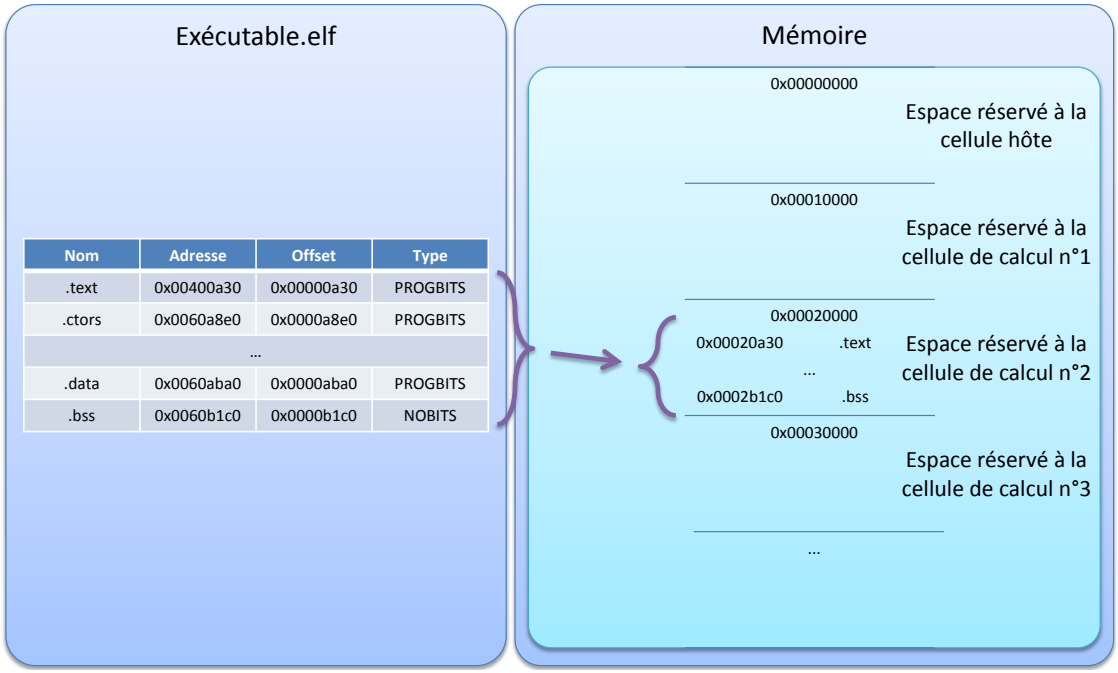


Figure 4.6 – *Exemple de placement des sections d'un fichier ELF en mémoire.*

transmet le pointeur vers l'adresse de début du noyau à la cellule hôte.

Du côté de la cellule hôte, après avoir reçu l'ordre d'exécution assorti du pointeur vers le noyau, le runtime récupère les paramètres transmis par le biais de l'interface, puis vient se brancher à l'adresse du noyau. Le noyau est donc exécuté, et les appels à des routines MPI sont transmis à la cellule hôte par le même mécanisme d'interface jusqu'à la fin de l'exécution.

4.1.3 Limitations de la plateforme SHP par rapport à SPoRE

Comme précisé précédemment, l'ambition de cette première plateforme est de valider l'architecture de SPoRE. Par conséquent, un certain nombre de spécifica-

tions n'ont pas été implémentées, car non nécessaires pour cette première étape. Il s'agit notamment de la reconfiguration matérielle, de l'utilisation des descripteurs d'application et du serveur de données.

Ainsi, la reconfiguration dans SHP se limite au code des noyaux exécutés sur les cellules de calcul. La reconfiguration dynamique partielle des ressources matérielles n'est ainsi pas supportée par cette plateforme. Néanmoins, notre principe de chargement du code du noyau par la cellule hôte permet une flexibilité de la plateforme, autorisant l'exécution de n'importe quel code logiciel. De plus, la séparation des cellules de calcul en entités indépendantes les unes des autres, aussi bien que vis-à-vis de la cellule hôte permet d'envisager l'utilisation de la RDP pour modifier le contenu de celles-ci ultérieurement. Ainsi, il suffira de rajouter un service de reconfiguration matérielle dans la cellule hôte afin de permettre la modification des PEs de manière dynamique.

D'autre part, la structure des applications destinées à SHP est relativement simple, celles-ci ne possédant que deux couches : la partie principale de l'application, qui s'exécute sur la cellule hôte, et les noyaux, compilés pour les cellules de calcul. À ce stade, il n'est pas nécessaire de posséder une couche de compatibilité. En effet, les cellules de calcul étant homogènes, la manière de dialoguer avec les noyaux est unique et ne nécessite pas d'être virtualisée. De ce fait, la structure de l'application séparée en plusieurs descripteurs, justement destinée à ajouter cette couche de virtualisation, n'est pas nécessaire.

Enfin, nous avons choisi de ne pas implémenter à ce stade la partie serveur de données. En effet, la centralisation des données constituant l'application est surtout destinée à permettre, pour chaque nœud, de ne télécharger que les implémentations des noyaux correspondant aux types des cellules locales. De plus, l'utilisation de MPI permet une centralisation automatique des résultats par échanges de messages, rendant inutile d'envoyer ceux-ci sur un serveur pour les rassembler. Nous avons donc jugé prématuré d'introduire l'élément serveur de données dans cette plateforme, celui-ci n'étant que peu utile dans ce cas précis. Sur cette plateforme, les noyaux sont donc stockés localement sur chaque nœud.

4.2 Test et validation de l'implémentation

Il convient maintenant de tester cette implémentation au moyen d'une application représentative. Il faudra notamment vérifier la validité du modèle des nœuds, structurés en cellules, en vérifiant la facilité de déploiement de l'application. Concernant les performances de l'application de test, celles-ci ne seront pas représentatives de la puissance d'un HPC : en effet, l'exécution sur des cellules de calcul à base de MicroBlazes ne permet pas de réaliser de hautes performances. En revanche, celles-ci pourront être comparées entre plusieurs configurations de la plateforme, permettant d'orienter les choix futurs sur cette base. Dans cette section, nous pré-

sentons dans un premier temps l'application choisie pour réaliser les tests, pour ensuite en étudier les résultats.

4.2.1 L'application de test

Afin de valider la plateforme SHP, nous devons sélectionner une application de test MPI. Celle-ci doit être de type HPC, afin de supporter la structure de la plateforme en nœuds distribués sur un réseau. Néanmoins, il n'est pas nécessaire qu'elle réalise des calculs intensifs, d'autant plus que le support d'exécution des noyaux est un processeur de faible puissance.

4.2.1.a Choix de l'application de test

La NASA, organisme spatial américain, s'intéresse beaucoup à la thématique du calcul parallèle, notamment via son département NASA Advanced Supercomputing (NAS). Celui-ci met notamment à disposition un ensemble de programmes destinés à tester les performances (« benchmarks ») des calculateurs parallèles : NAS Parallel Benchmark (NPB) [70, 4]. NPB est composé de 11 benchmarks, proposant ainsi différents critères de test à l'utilisateur. Cet ensemble de benchmarks, proposé dans sa première version en 1992, est maintenant largement utilisé pour tester des plateformes parallèles. La première version de NPB était uniquement un ensemble de spécifications algorithmiques, l'implémentation du test étant laissée à la discrétion de l'utilisateur. Par la suite, ces tests ont été fournis en MPI, puis enfin en utilisant d'autres implémentations.

Chaque test de la suite NPB propose différentes classes de calcul, correspondant à différentes quantités de données à traiter. Initialement disponibles avec deux classes, A et B, ainsi qu'une version simplifiée non destinée à des fins de benchmarking, la classe S, ces tests se sont enrichis au fur et à mesure de l'évolution de la puissance des HPCs. C'est ainsi que la dernière version de la suite, NPB 3.2, propose des implémentations MPI, OpenMP, Java, et HPF des tests, pour des classes allant jusqu'à F. Une version multi-zones de certains benchmarks est également disponible, combinant MPI et OpenMP pour tester la capacité d'un système à tirer parti des différents niveaux de parallélisme.

Étant donné la stabilité de ces tests et leur large diffusion, nous avons décidé de choisir l'un d'entre eux pour valider notre plateforme. Comme indiqué précédemment, notre but n'est pas de tester la puissance de calcul de notre plateforme, basée sur des processeurs peu puissants. En revanche, nous devons valider le principe choisi pour la communication, et vérifier que l'architecture passe correctement à l'échelle, en multipliant les nœuds et les cellules de calcul.

Nous avons donc choisi le test Integer Sort (IS). Il s'agit d'un algorithme de tri-casier (« bucket sort ») distribué utilisant MPI pour la communication entre les nœuds. Le contenu de celui-ci est présenté en algorithme 1.

```
Données : Soient P processeurs, C casiers et V valeurs à trier
pour tous les processeurs faire
    pour i itérations faire
        pour chaque valeur faire
            Placer dans le casier correspondant au rang;
        fin
        Additionner le nombre de valeurs dans les C casiers sur la totalité des
        P processeurs;
        numéro_proc = 0;
        casier_courant = 0;
        pour chaque processeur faire
            valeurs_envoyées = 0;
            tant que valeurs_envoyées inférieur à V/P faire
                Envoyer le contenu sur le processeur numéro_proc;
                ajouter compter_valeurs(casier_courant) à valeurs_envoyées;
                incrémenter casier_courant;
            fin
            incrémenter numéro_proc;
        fin
        Créer un casier pour chaque valeur;
        Compter le nombre d'occurrence de chaque valeur;
    fin
fin
```

Algorithme 1: Algorithme de tri de NPB IS.

Le principe du tri-casier consiste à disposer de plusieurs casiers, chacun correspondant à un intervalle de valeurs. Chaque entier est envoyé vers le casier correspondant à sa valeur, puis chaque casier est trié à son tour. En utilisant une structure parallèle, chaque nœud génère au démarrage un vecteur de nombres entiers aléatoires, qui sont distribués parmi les nœuds, chacun correspondant à un casier.

Le cœur du calcul n'est donc pas très intensif, consistant à tester vers quel nœud doit être envoyé chaque entier, puis à trier les casiers locaux. En revanche, l'aspect communication est relativement développé, car chaque nœud communique potentiellement avec tous les autres, afin de leur transmettre les valeurs correspondant à leur intervalle.

4.2.1.b Adaptation de l'application à la plateforme

L'application NPB IS, dans sa version MPI, est constituée d'un simple fichier de code en C. Celui-ci est constitué de trois sections principales : une pour l'initialisa-

tion, une pour le tri, puis une pour la vérification. L'intérêt du test résidant dans le cœur du tri, qui contient la communication MPI, nous considérons les parties initialisation et vérification comme du contrôle, qui sera donc réalisé sur la cellule hôte. Ainsi, seul l'algorithme de tri est réalisé sur les cellules de calcul, évitant d'allonger la durée des tests en exécutant les fonctions attenantes sur les cellules de calcul. En effet, ces deux parties d'initialisation et de vérification ne font pas partie du test à proprement parler : celles-ci n'existeraient pas dans une application réelle, et ne sont d'ailleurs pas prises en compte dans la durée du test de l'application initiale.

Le cœur de l'application, c'est-à-dire l'algorithme de tri, est remplacé par un proxy dans l'application hôte. On crée simultanément une application de type MicroBlaze, dans laquelle ce code est placé, et à laquelle est également adjoint un proxy qui permettra de transmettre les paramètres en provenance de la cellule hôte. La communication initiée par le noyau sur une cellule de calcul sera quant à elle transmise à la cellule hôte via le proxy.

On dispose ainsi de la répartition déjà explicitée de l'application, avec la communication entre les cellules passant par la cellule hôte. L'application NPB IS dispose d'éléments permettant de mesurer la durée de l'application (« timers »), et notamment les temps de calcul et de communication. Ceux-ci ne faisant pas partie de l'algorithme de tri, ils seront exécutés sur la cellule hôte. Les temps d'exécution correspondent au temps entre le déclenchement de l'exécution d'un noyau et le retour indiquant que celle-ci est terminée. Les durées de communication seront mesurées entre la requête d'une cellule de calcul et la réponse de la cellule hôte à cette requête. Les communications étant bloquantes, l'exécution du noyau est figée durant les communications. Le timer correspondant au temps de calcul sera alors suspendu durant les appels de communication.

4.2.2 Évaluation des performances

Nous avons donc utilisé le benchmark NPB IS pour valider la plateforme SHP. Afin de tenter d'identifier d'éventuels facteurs limitants, nous avons réalisé plusieurs configurations de test dans le but de les comparer entre elles. Ceci nous permettra d'identifier les facteurs influençant le plus les résultats.

4.2.2.a Configuration des tests

L'exécution de l'application, compilée avec la bibliothèque MPICH, est réalisée avec l'ordonnanceur associé MPD (MultiProcessing Daemon). Celui-ci, lancé sur tous les nœuds ainsi que sur le nœud maître, permet de relier l'ensemble des composants de SPoRE dans un réseau virtuel ayant une topologie en anneau. Par la suite, le lancement d'une application MPI sur le nœud maître utilisera ce réseau pour distribuer les travaux (*jobs* MPI). Sur chaque nœud, on autorise un nombre de jobs égal. L'application nécessitant que tous les processus soient exécutés simulta-

nément, le nombre de job sur chaque nœud doit être inférieur ou égal au nombre de cellules locales, afin de permettre à notre couche de compatibilité de les distribuer sur les cellules de calcul.

Comme indiqué auparavant, on dispose de deux implémentations de SHP, selon que les cellules de calcul disposent ou non de cache. Ce premier paramètre nous autorise deux types de tests, mais autorise un nombre de cellules différent pour ces deux configurations. On dispose en tout de 13 cartes ml507, pouvant donc contenir chacune 3 à 7 cellules selon la configuration du cache. À la lumière des premiers tests, nous avons choisi de toujours désactiver le cache de données, celui-ci influant négativement sur les résultats. La figure 4.7 montre la comparaison entre l'exécution de l'application avec les deux caches activés et désactivés. On voit clairement que

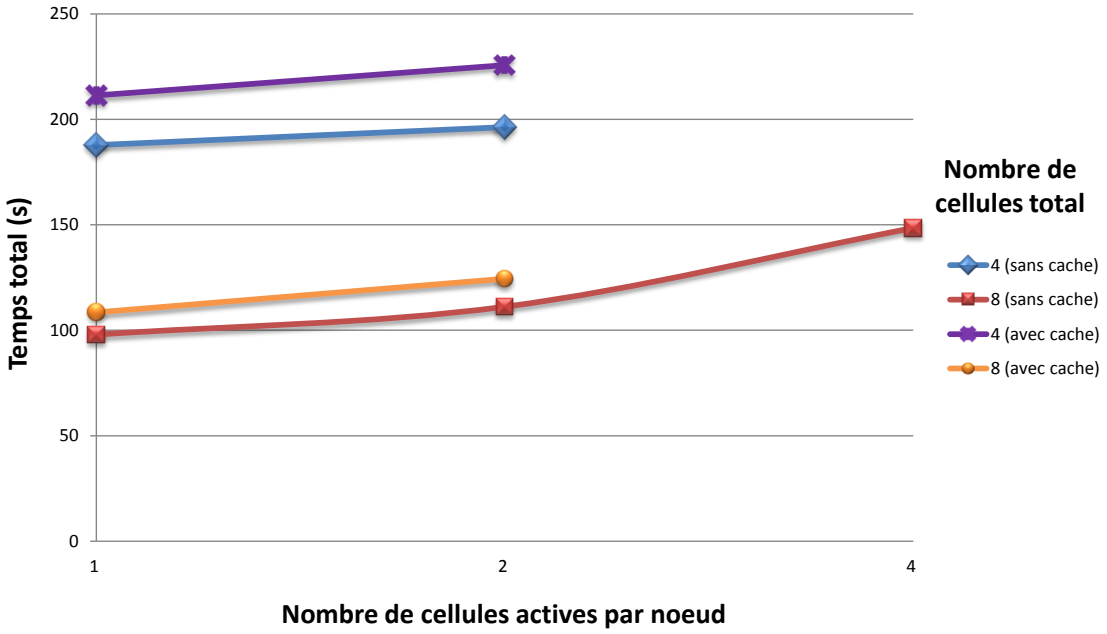


Figure 4.7 – *Évolution de la durée passée à communiquer par rapport à la concentration des cellules.*

l'activation des deux caches mène à des temps d'exécution plus longs, c'est-à-dire de moindres performances.

En effet, en raison de la nature particulière de l'algorithme, et notamment de son accès désordonné aux données, le processeur passe beaucoup de temps à mettre à jour le cache. Les résultats présentés par la suite, et indiquant si le cache est activé ou non, ne concernent donc que le cache d'instruction, celui de données étant toujours désactivé. Néanmoins, l'utilisation du seul cache d'instruction nécessite quand même le recours à deux canaux mémoire sur les cellules concernées.

Un autre paramètre de configuration disponible est de faire varier le nombre de cellules participant au calcul, afin de modifier la charge de travail de chacune. On peut ainsi faire varier le nombre de cellules disponibles dans l'ensemble de la

plateforme, à nombre de cellules par nœud constant. Pour ce test, on définira un nombre de cellules par nœud fixe, et on connectera plus ou moins de nœuds sur la plateforme.

Enfin, une dernière possibilité est de conserver un nombre de cellules constant au travers de toute la plateforme, mais de faire varier sa répartition sur les nœuds. Ainsi, pour n cellules disponibles dans la plateforme, on pourra les distribuer sur m nœuds, à raison de $\frac{n}{m}$ cellules par nœud. En faisant varier m, on obtiendra ainsi une comparaison des performances de communication locales par rapport aux communications distantes.

En jouant sur ces trois facteurs, nous avons réalisé différents tests sur la plateforme SHP.

4.2.2.b Résultats

En fonction de la quantité de PEs disponibles, nous avons choisi d'utiliser le test NPB IS dans sa classe A. Celle-ci, bien que peu adaptée aux supercalculateurs contemporains, reste représentative d'un HPC, contrairement à la classe S, qui n'est utilisée que pendant la mise en place du benchmark, pour des résultats rapides. Face au faible nombre de PEs dans notre architecture, moins d'une centaine, une classe supérieure ne serait pas adaptée. La limitation provient notamment de la mémoire qui, avec 256 Mio par nœud, ne peut contenir une quantité de données supérieure à celle de la classe A.

Nous avons choisi de réaliser des tests avec un nombre de cellules totales de 4, 8, 16 et 32, et pour un nombre de cellules par nœud de 1, 2 et 4. Ce sous-ensemble des possibilités disponibles permet une bonne vue d'ensemble du passage à l'échelle de la plateforme, en montrant l'évolution des performances à chaque doublement de la capacité. On différenciera dans les tests le temps passé par l'application à calculer de celui passé à communiquer, chacun nous donnant des informations différentes.

Temps de communication

La figure 4.8 présente les résultats obtenus sur le temps de communication, en fonction du nombre de jobs exécutés sur chaque nœud, à nombre total constant. On remarque que lorsqu'on augmente la concentration des jobs sur un même nœud, le temps de communication augmente fortement. Par exemple, pour 8 jobs exécutés sur des cellules sans cache, le temps passé par l'application varie de moins de 10 s lorsqu'ils sont exécutés chacun sur un nœud, à plus de 30 s lorsqu'on en rassemble 4 par nœud. Le même constat est également visible pour les cellules avec cache. On remarque également que l'activation du cache d'instruction accélère les opérations, ce qui est le comportement correct attendu, contrairement à l'activation du cache de données.

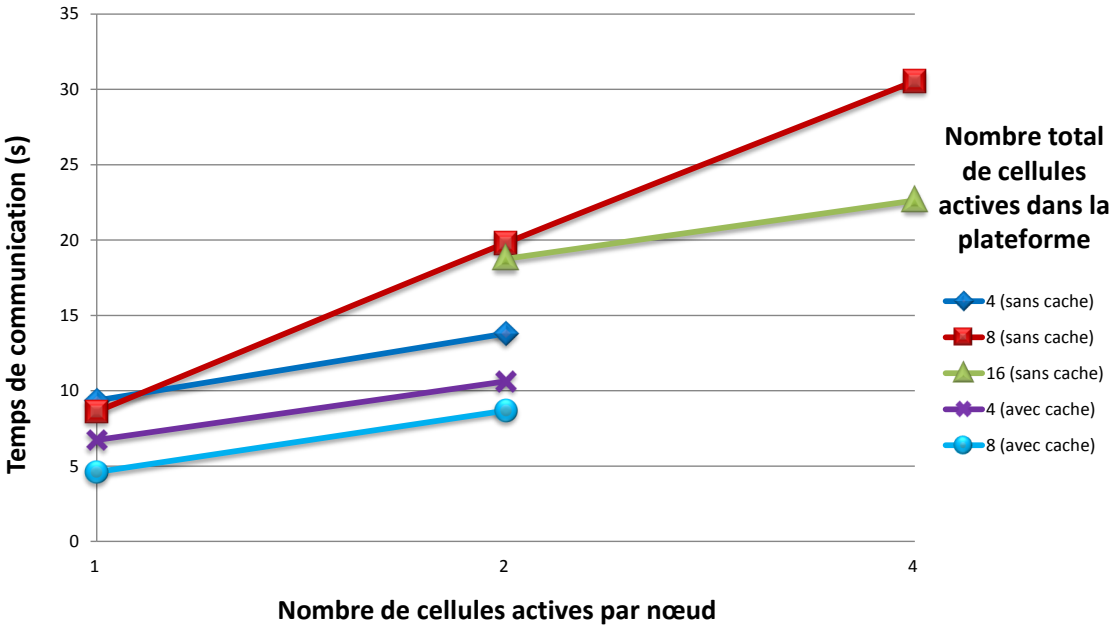


Figure 4.8 – *Évolution de la durée passée à communiquer par rapport à la concentration des cellules.*

L'augmentation du temps de communication avec la concentration des jobs peut sembler contre-intuitive. En effet, la communication via le réseau est généralement plus lente que la communication locale. Cela est dû à l'absence d'implémentation d'un protocole d'échange de données propre au SMP, tel OpenMP. En effet, utilisant MPI également pour la communication locale, les données doivent être copiées entre les plages mémoires pour être transférées. Or, toutes les cellules utilisant la même mémoire, un engorgement a lieu lors des accès à la mémoire, ce qui produit ce résultat.

Temps de calcul

Sur la figure 4.9, on présente les temps de calcul obtenus pour différents nombres de jobs, à nombre de cellules par nœud constant. On constate bien l'effet recherché, à savoir qu'une plus grande distribution résulte en un temps d'exécution plus réduit. Le parallélisme prend alors tout son sens, l'effort conjoint de plusieurs cellules étant plus efficace qu'en nombre réduit.

Entre les différentes courbes, on retrouve le résultat précédent, dans lequel une plus grande concentration des cellules produit un temps plus élevé, en raison de la congestion. En revanche, cette différence est bien plus réduite que sur le temps de communication, et n'est réellement appréciable que lorsque 4 cellules sont actives sur chaque nœud.

Concernant l'activation du cache d'instructions, cette fois-ci, on note un très

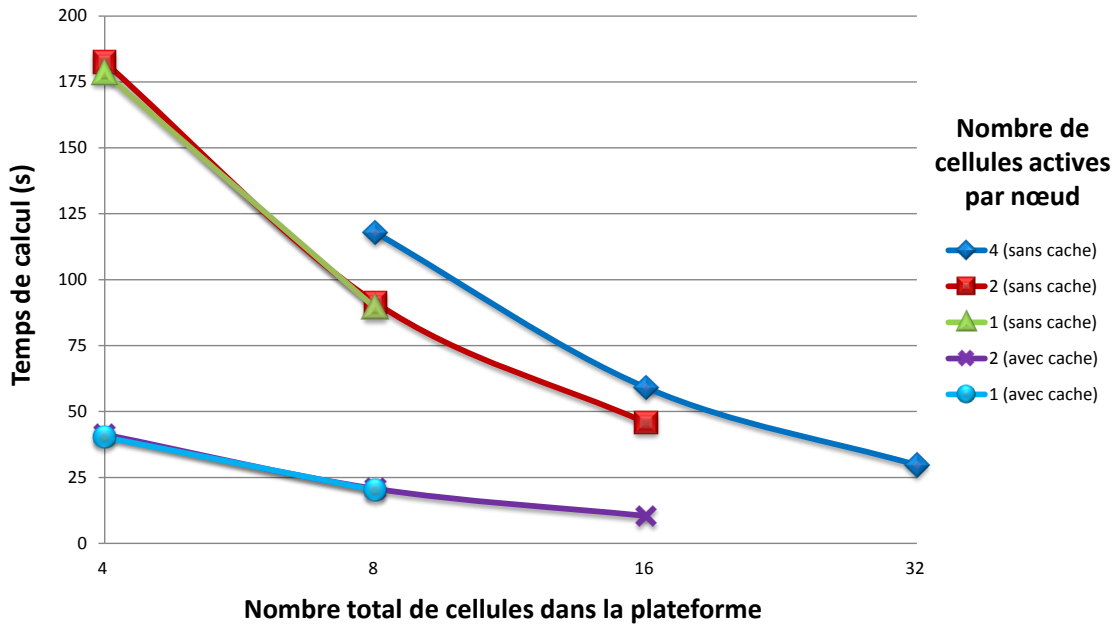


Figure 4.9 – *Évolution de la durée des calculs par rapport au nombre total de cellules actives dans la plateforme.*

gros gain de temps. En effet, sur la partie calcul, sa désactivation nécessite un accès mémoire constant pour aller récupérer les instructions. Or, cette opération, déjà naturellement plus lente qu'un accès local, dépend également de la concurrence sur la mémoire partagée.

Temps total

Enfin, la figure 4.10 présente les temps globaux d'exécution de l'application, contenant à la fois le calcul et les communications. Bien que les communications soient ralenties par la concentration des jobs, on remarque que cela a une influence relativement réduite sur la totalité du temps d'exécution. En effet, celle-ci n'est réellement sensible que lorsque l'on concentre quatre cellules actives sur le même nœud. Dans ce cas, la concurrence sur l'accès à la mémoire est alors très marquée, à la fois sur les communications et sur le calcul. En revanche, la tendance correcte, consistant à réduire le temps de calcul en augmentant le nombre de nœuds est toujours visible, et ce quelle que soit la concentration.

4.3 Conclusion sur la plateforme SHP

De l'exécution du benchmark NPB IS sur la plateforme SHP, on peut tirer plusieurs résultats. Le premier est un résultat positif : nous avons été capables, avec peu de modifications, d'exécuter une application MPI. En effet, la seule adaptation

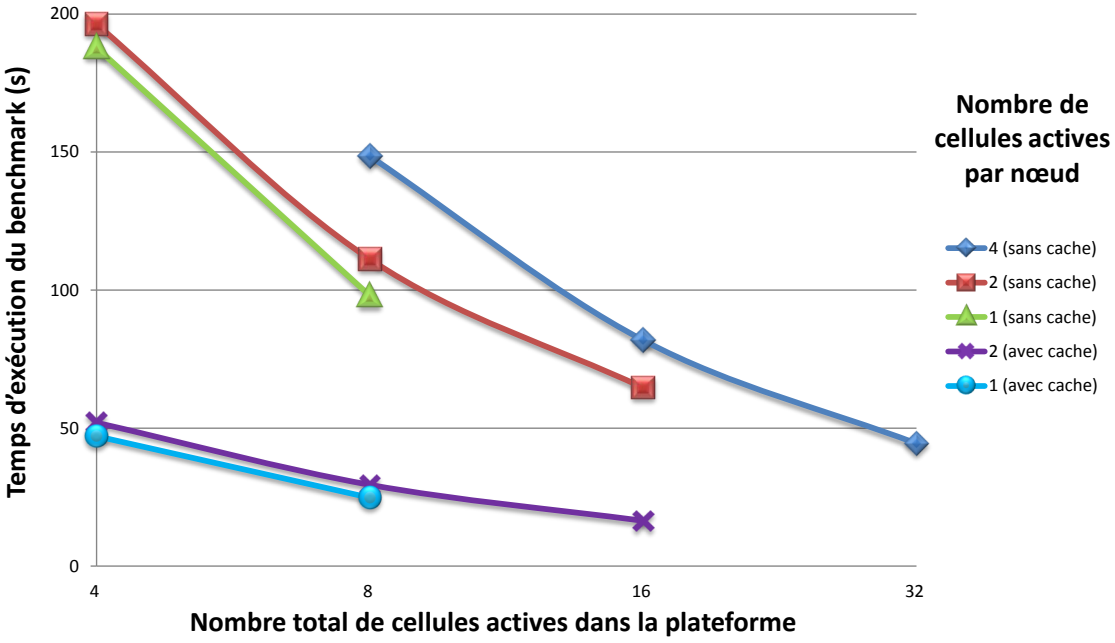


Figure 4.10 – *Évolution de la durée totale du benchmark par rapport au nombre total de cellules actives dans la plateforme.*

qui a été nécessaire sur cette application a consisté à écrire la couche de proxy permettant de transférer les appels aux noyaux vers les cellules de calcul. Or, ce travail est aisément automatisable. On pourrait tout à fait imaginer introduire des pragmas indiquant quelle fonction est un noyau, puis exécuter un pré-compilateur qui séparerait ces fonctions des autres, et mettrait en place le proxy sur la seule base des paramètres de la fonction.

Par ailleurs, nous avons également constaté un problème au niveau de la communication par la mémoire partagée. Comme indiqué, celui-ci est principalement du à la manière dont nous avons implémenté la communication. En effet, plusieurs copies concurrentes entre plages d'une même mémoire sont naturellement ralenties par l'attente mutuelle. Ce problème pourrait être résolu par l'utilisation d'une mémoire locale pour les données dans chaque cellule de calcul combiné à un bus d'échange direct entre les cellules. À condition que les échanges entre les cellules ne partagent pas un même canal, par exemple par l'utilisation d'un bus de type crossbar ou d'un NoC, on éviterait alors la concurrence sur les communications. Il faudra donc, pour la plateforme finale, abandonner le principe du SMP, pour adopter une topologie plus proche du NoC.

Néanmoins, le point principal était ici de valider la structure générale de SPoRE, ce qui a été réalisé avec succès. Nous pouvons donc maintenant nous concentrer sur l'aspect matériel, et notamment sur l'utilisation de la RDP pour les cellules de calcul.

Chapitre 5

Seconde implémentation : plateforme matérielle reconfigurable

Sommaire

À la suite de l'implémentation SHP de SPoRE, nous avons mis en place une deuxième implémentation, cette fois-ci axée sur la reconfiguration dynamique et les PEs matériels. Cette seconde implémentation bénéficie de la base architecturale validée sur SHP mais, différant de celle-ci au niveau des objectifs, se verra adaptée pour ceux-ci.

Si la première implémentation était basée sur MPI et la communication directe entre noyaux, nous avons ici choisi de nous concentrer sur l'aspect flot de données (« streaming »). Cette plateforme n'intègre donc pas MPI, car bien que cette interface soit très répandue dans le logiciel, c'est encore loin d'être le cas pour les PEs matériels. L'un des points principaux de cette implémentation sera l'intégration de la reconfiguration matérielle, utilisant des fichiers de configuration pour instancier les PEs à la volée.

Pour ces raisons, nous avons choisi le nom de Hardware Stream Dynamic Platform (HSDP) pour cette seconde implémentation de SPoRE. Cette fois-ci, on retrouvera dans la plateforme le serveur de données, qui permettra entre autres de stocker les implémentations requises par chaque nœud, ainsi que les descripteurs permettant la représentation de l'application.

Ce chapitre est organisé comme suit : dans un premier temps, nous présentons l'implémentation en elle-même en section 5.1, son architecture matérielle et ses composantes logicielles. Ensuite, en section 5.2, nous décrivons la syntaxe des descripteurs, qui sont implémentés en XML. Enfin, dans la section 5.3, nous détaillons les tests soumis à cette plateforme et leurs résultats.

5.1 Caractéristiques de l'implémentation

L'implémentation HSDP de SPoRE conserve bien entendu l'architecture des nœuds en cellules, avec la cellule hôte servant à la fois de passerelle entre le nœud et le reste du réseau SPoRE, et prenant en charge la supervision du nœud : ordonnancement, reconfiguration, etc. Dans SHP, la supervision de la cellule hôte n'a nécessité que peu de développement, la partie ordonnancement étant déjà prise en charge par MPICH. L'absence de celui-ci dans HSDP nécessitera de prendre en charge de bout en bout la gestion de l'application.

Afin de valider la partie reconfiguration matérielle de SPoRE, cette implémentation est tournée vers les PEs matériels. Dans l'absolu, il est toujours possible d'implémenter un PE logiciel sur une cellule de calcul pour y exécuter un noyau. Néanmoins, on ne mettra pas en place cette technique dans cette plateforme, car cela n'apporte rien sur la validation de la reconfiguration. Afin de conserver tout de même une compatibilité avec les noyaux logiciels, on autorise leur exécution sur le processeur de la cellule hôte.

5.1.1 Architecture matérielle

Comme indiqué, l'architecture matérielle d'un nœud HSDP conserve la répartition cellule hôte/cellules de calcul. En revanche, l'architecture de communication entre les cellules sera modifiée pour mieux tenir compte de la nature matérielle des PEs.

Du coté de la cellule hôte, on conservera une architecture proche de celle de SHP. En effet, cette architecture a prouvé qu'elle était à même de prendre en charge la gestion du nœud. On n'apportera donc que peu de modification sur l'architecture matérielle de cette cellule, et les principales modifications se situeront au niveau logiciel. En termes matériel, il faudra principalement ajouter le support de la reconfiguration.

En revanche, les cellules de calcul, elles, vont changer du tout au tout. En lieu et place d'un PE logiciel, on intègrera dans ces cellules une zone reconfigurable qui permettra d'implémenter divers PEs à l'exécution.

5.1.1.a Architecture de la cellule hôte

La base architecturale de la cellule hôte, bâtie autour d'un PowerPC, n'est pas remise en cause car elle a prouvé son bon fonctionnement sur la première plateforme. On présente celle-ci sur la figure 5.1.

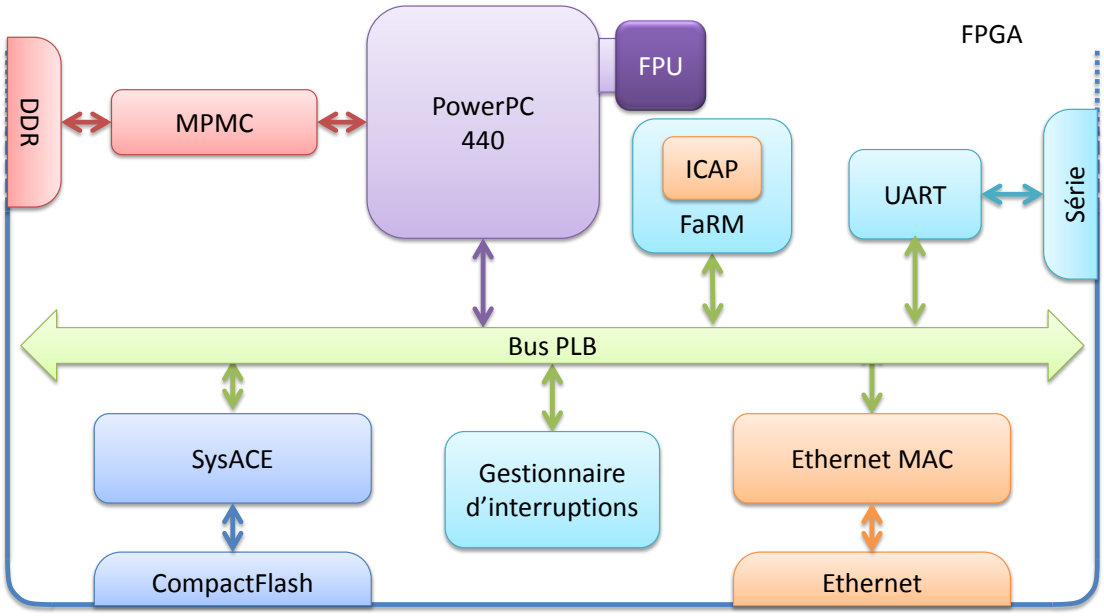


Figure 5.1 – *Architecture de la cellule hôte sur la plateforme HSDP.*

Le principal changement matériel de la cellule hôte concernera l'intégration d'un contrôleur de reconfiguration : FaRM. Ce composant permet de procéder à des

reconfigurations partielles, et sera accessible directement depuis Linux en écrivant les pilotes adéquats.

Autre changement, on adjoint une FPU au processeur PowerPC. Implémenté sur ressources reconfigurables, cet élément se couple au PowerPC et lui ajoute ainsi une capacité de calcul relativement élevée concernant les nombres décimaux. Le choix d'ajouter cet élément résulte du fait que les noyaux de calcul logiciels sont autorisés à s'exécuter sur la cellule hôte, et peuvent avoir besoin d'une forte capacité de calcul. Le reste des IPs composant la cellule hôte reste inchangé.

5.1.1.b Architecture des cellules de calcul

Dans cette implémentation, les cellules de calcul sont destinées à accueillir dynamiquement des PEs matériels. Pour cela, il faut réaliser une première division de la cellule en deux parties : une zone dynamique, où le PE sera placé par reconfiguration, et une zone statique, en charge de la gestion de la cellule. Le *contrôleur de noyau* représente la zone statique, tandis que l'*hôte de noyau* représente la zone dynamique, comme indiqué sur la figure 5.2.

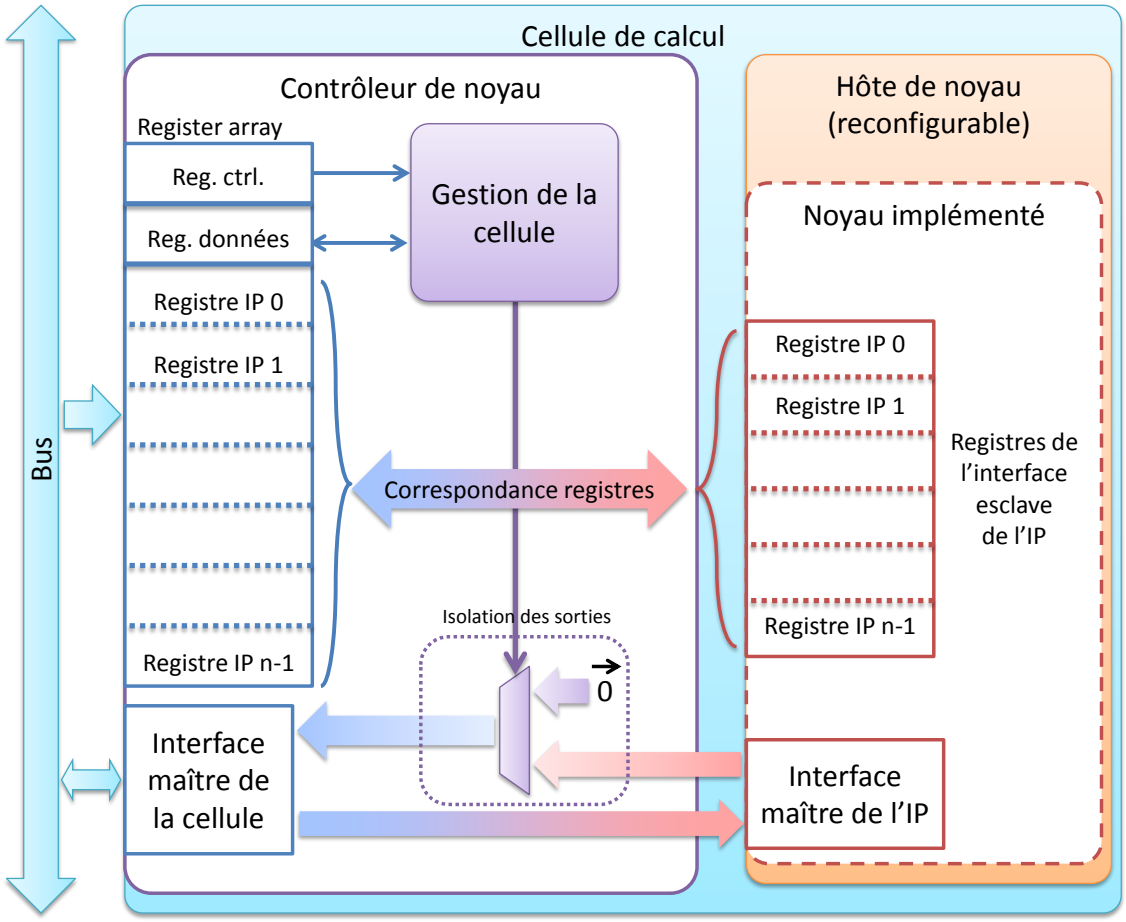


Figure 5.2 – *Architecture d'une cellule de calcul sur la plateforme HSDP.*

La principale attribution du contrôleur de noyau est technique. En effet, le processus de reconfiguration d'une zone peut produire des signaux aléatoires et perturber le système complet. Or, si les signaux aléatoires internes n'ont pas d'effet sur le reste du système, il existe des signaux de sortie en direction du bus : les signaux de l'interface maître (si elle existe) et ceux utilisés par l'interface esclave pour répondre aux requêtes de l'arbitre de bus. La présence de données aléatoires sur ceux-ci peut avoir pour effet de produire des requêtes incontrôlées et vides de sens, qui dans le pire des cas peuvent placer l'arbitre dans un état indéterminé, provoquant l'arrêt de son fonctionnement.

De ce fait, le contrôleur de noyau doit veiller à ce que pareille situation ne se produise pas. Pour cela, tous les signaux de sortie de l'hôte de noyau sont isolés du bus par le biais du contrôleur de noyau, qui permet de forcer ceux-ci à l'état logique bas lors de la phase de reconfiguration.

Outre cette fonction, le contrôleur de noyau permet d'intégrer différents éléments de mesure. Par exemple, il est possible d'ajouter un timer matériel permettant de mesurer la durée d'exécution du noyau au cycle près.

En sus des registres de l'IP implémenté sur l'hôte de noyau, le contrôleur de noyau nécessite également des registres pour permettre sa propre configuration. En raison de cela, les registres du noyau sont décalés d'une constante correspondant au nombre de registres dédiés au contrôleur de noyau. Ce décalage sera automatiquement géré par runtime afin d'être invisible au développeur de l'application.

Nous avons choisi d'utiliser un nombre de registres fixe : un registre de contrôle et un registre de données. À l'aide de ces deux registres, interagir avec le contrôleur de noyau est réalisé en deux étapes. La première étape consiste à écrire une commande dans le registre de contrôle. L'écriture de cette commande configure le comportement du registre de données, et les prochaines interactions avec celui-ci seront relatives à cette commande jusqu'à la prochaine écriture dans le registre de contrôle. Par exemple, après avoir configuré le registre de contrôle pour la gestion de la connexion de l'hôte de noyau, les prochaines écritures dans le registre de données commanderont l'isolation des signaux sortant de la zone reconfigurable. On ajoute également la possibilité de venir lire le registre de contrôle, opération qui retourne l'état interne du contrôleur de noyau : état de la connexion et présence ou non du timer.

5.1.1.c Communication intra-nœud

Une architecture axée flot de données utilise la mémoire différemment d'une architecture basée sur MPI. Dans ce genre d'architecture, les PEs disposent en général d'une petite mémoire locale, dans laquelle sont stockées les données à traiter. L'accès à cette mémoire peut se faire par le biais d'un espace d'adressage sur un bus, dans lequel les données sont écrites. De la même manière, la récupération des données de sortie est faite par lecture dans une plage d'adresses, qui peut

éventuellement être différente de la plage d'entrée.

Certains IPs proposent une interface maître, qui peut se charger d'aller récupérer les données en mémoire directement à l'aide d'un gestionnaire d'accès direct à la mémoire (Direct Memory Access – DMA). Dans ce cas, il suffit de fournir à l'IP l'adresse où les données sont stockées, leur taille, et l'adresse à laquelle stocker les résultats. L'approvisionnement de l'IP est alors géré automatiquement, ainsi que l'écriture des résultats.

Dans le cas où plusieurs IPs sont chaînés, c'est-à-dire lorsque les données de sortie d'un PE sont utilisées en entrée par un autre, il est parfois possible de passer outre le stockage en mémoire intermédiaire, les données étant directement transférées de l'un à l'autre. Cette opération peut s'effectuer via le bus ou, dans le cas de systèmes plus spécifiques, via une connexion directe entre les deux IPs.

Dans notre cas, on cherche à rester le plus générique possible. Ainsi, pour chaque cellule de calcul, les deux interfaces, maître et esclave, sont fournies. En fonction du noyau qui sera présent dans la cellule, celui-ci pourra ainsi utiliser ou non l'interface maître. On présente les branchements de ces deux interfaces sur la figure 5.3.

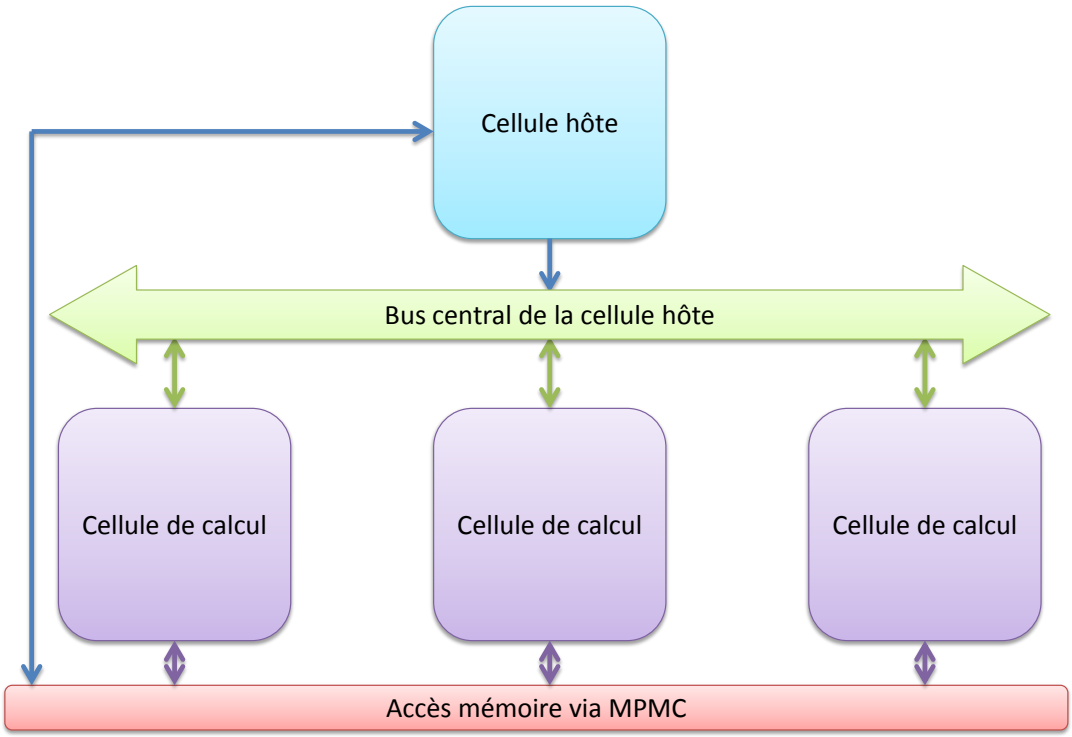


Figure 5.3 – *Canaux de communication au sein de la plateforme HSDP.*

Les interfaces esclaves des cellules de calcul sont branchées sur le bus de la cellule hôte, sur lequel cette dernière dispose d'un accès maître. Ainsi, la cellule hôte peut accéder aux registres de chaque cellule de calcul. Cette méthodologie ressemble au principe des MailBoxes sur la première plateforme, mais permet un

adressage par registres et plages mémoire, plus adapté aux IPs matériels.

Concernant l'interface maître des cellules de calcul, deux possibilités sont présentes : la brancher sur ce même bus central et permettre l'accès à la mémoire via le bus, ou bien effectuer un accès direct à la mémoire par le MPMC. Dans les deux cas, un arbitre permet de partager l'accès à la mémoire : celui du bus dans le premier cas, et celui du MPMC dans le second. Nous avons choisi de brancher l'interface maître directement sur la mémoire via le MPMC. Ainsi, on évite d'encombrer le bus par les accès mémoire : celui-ci restera disponible pour la cellule hôte quand les cellules de calcul accèderont à la mémoire.

5.1.1.d Occupation des ressources matérielles

L'occupation en ressources matérielles des cellules de calcul est moins représentative sur cette plateforme que sur la précédente. En effet, celle-ci dépend en grande partie de la taille de l'hôte de noyau. Or, cette taille est arbitraire, et doit être choisie en fonction de l'occupation des PEs destinés à y être implémentés. Il est donc possible, pour des nœuds disposant du même nombre de cellules, d'obtenir des tailles très différentes en fonction de l'espace alloué à l'hôte de noyau. Les tailles des éléments statiques sont présentées dans la table 5.1.

Resource	Cellule hôte	Gestionnaire de noyau
LUTs	7 658 (17,1%)	708 (1,6%)
BRAMs	10 (6,8%)	0 (0,0%)
DSP48Es	13 (10,2%)	0 (0,0%)

Tableau 5.1 – *Ressources utilisées par les éléments statiques de la plateforme HSDP. Les pourcentages indiquent l'occupation du composant Virtex 5 fx70t.*

On constate, par rapport à la table 4.1, une forte augmentation de la consommation en LUTs de la cellule hôte par rapport à la plateforme SHP. Cela est principalement du à l'ajout de la FPU : implémenté de manière reconfigurable, cet élément se couple au PowerPC et lui ajoute ainsi une capacité de calcul relativement élevée concernant les nombres décimaux. Or, cet élément est complexe et monopolise ainsi près de la moitié des ressources en LUTs de la cellule hôte.

De leur coté, les cellules de calcul contiennent une partie statique, le contrôleur de noyau, présenté dans ce tableau avec le timer intégré. On se rend compte que la taille totale de cette partie statique est relativement réduite. Bien entendu, cette taille est susceptible de varier si l'on ajoute d'autres composants de mesure ou de contrôle du noyau. On ne peut donner d'estimation de la taille de l'hôte de noyau à ce stade, car celui-ci dépendra de l'utilisation que l'on souhaite en faire, qui déterminera son dimensionnement.

5.1.1.e Bilan matériel

Dans cette seconde plateforme, la partie matérielle a nécessité un travail de développement plus conséquent que la première plateforme. En effet, l'utilisation de la RDP dans un système nécessite à l'heure actuelle un développement particulier pour permettre une utilisation générique. C'est justement une des raisons qui nous ont poussés à réaliser ce travail. En utilisant les cellules de calcul décrites dans cette section, nous sommes maintenant à même de placer des IPs dynamiquement dans notre système.

5.1.2 Architecture logicielle

Pour cette plateforme, nous avons besoin d'un environnement logiciel bien plus développé que pour la précédente. En effet, en l'absence de MPI, il va falloir gérer l'ordonnancement, mais également prendre en charge la communication entre les nœuds. De plus, nous avons ajouté le serveur de données, avec lequel il faudra communiquer pour télécharger les données. Enfin, afin de tirer parti de la reconfiguration matérielle, il sera nécessaire de disposer d'un service en charge de cette reconfiguration. De plus, dans cette plateforme, nous avons ajouté deux nouveaux éléments : l'ordonnanceur global, situé sur le nœud maître, ainsi que le serveur de données sont des nœuds spéciaux.

5.1.2.a Ordonnanceur global

L'ordonnanceur global constitue à la fois le point d'entrée du réseau SPoRE et le lien avec l'utilisateur. En effet, c'est à cet ordonnanceur que l'utilisateur doit soumettre les applications à exécuter. De leur côté, tous les autres nœuds, y compris les serveurs de données, se connectent au nœud maître au démarrage, afin d'être référencés dans le réseau SPoRE.

Le principe de l'inscription des nœuds permet de disposer d'une structure évolutive, dans laquelle il est possible de rajouter des ressources de calcul à la volée. Il est également possible pour un nœud de quitter ce réseau pour, par exemple, se connecter à un autre. Comme nous ne prenons pas en compte dans ces travaux de recherche la gestion des erreurs, la procédure à suivre pour quitter le réseau SPoRE est définie : un nœud souhaitant quitter le réseau signifie qu'il ne veut plus que lui soit dévolu de nouveaux travaux. Lorsqu'il a terminé la totalité des travaux qui ont été soumis avant cette requête, il est alors autorisé à quitter le réseau. Cette solution évite de devoir prendre en charge la migration d'un travail non terminé, ce qui peut être assimilé à de la non-préemption.

Lors d'une demande d'exécution de l'utilisateur, l'ordonnanceur obtient auprès d'un serveur de données le descripteur d'application à partir de la référence fournie par l'utilisateur. L'ordonnanceur partitionne le travail et le distribue sur les diffé-

rents nœuds. Pour cela, l'ordonnanceur crée autant de descripteurs d'applications que de partitions, formant ainsi des sous-applications qui pourront être attribuées aux nœuds.

On souhaite à terme intégrer un certain nombre de métriques pour les choix de l'ordonnanceur global en termes de répartition des travaux. Par exemple, en fonction de la topologie du réseau : on privilégiera les nœuds entre lesquels le débit est élevé pour distribuer des portions de l'application comportant des communications importantes en quantité de données.

5.1.2.b Serveur de données

Le serveur de données est un autre nœud spécial dont le but est de fournir et récupérer des données vers et depuis les nœuds de calcul. Plusieurs serveurs de données peuvent coexister dans un même réseau SPoRE. Dans une première approximation, on supposera qu'ils fonctionnent en miroir, c'est-à-dire que leur contenu est à tout moment identique. D'autres modèles de contenu pourraient être envisagés, tels qu'un serveur central contenant la liste des fichiers et leur localisation sur les serveurs de données, mais ceci sort du cadre de notre étude.

L'idée du fonctionnement en miroir permet à chaque nœud de calcul se connectant à un réseau SPoRE de recevoir de la part de l'ordonnanceur global l'adresse d'un serveur de données en fonction de la topologie du réseau. Par exemple pour un réseau à deux zones principales reliées par un lien à faible débit, on pourra souhaiter placer un serveur dans chacune de ces zones pour éviter ce lien.

5.1.2.c Environnement d'exécution des nœuds

Sur les nœuds de calcul, nous avons mis en place un runtime qui permet la gestion du nœud depuis la cellule hôte. Cet environnement d'exécution est composé des services suivants, comme présenté sur la figure 5.4.

- Un ordonnanceur local
- Un gestionnaire de mémoire (Lite Memory Manager – LMM)
- Un gestionnaire de reconfiguration
- Un gestionnaire de stockage
- Un gestionnaire de cellules
- Un service de mesure de durée (« timer »)
- Un parser XML

Ordonnanceur local

L'ordonnanceur local est le point d'entrée du runtime d'un nœud. En effet, au démarrage du nœud, celui-ci se connecte à l'ordonnanceur global, qui pourra alors à partir de ce moment lui soumettre du travail. Lors de la connexion à l'ordonnanceur

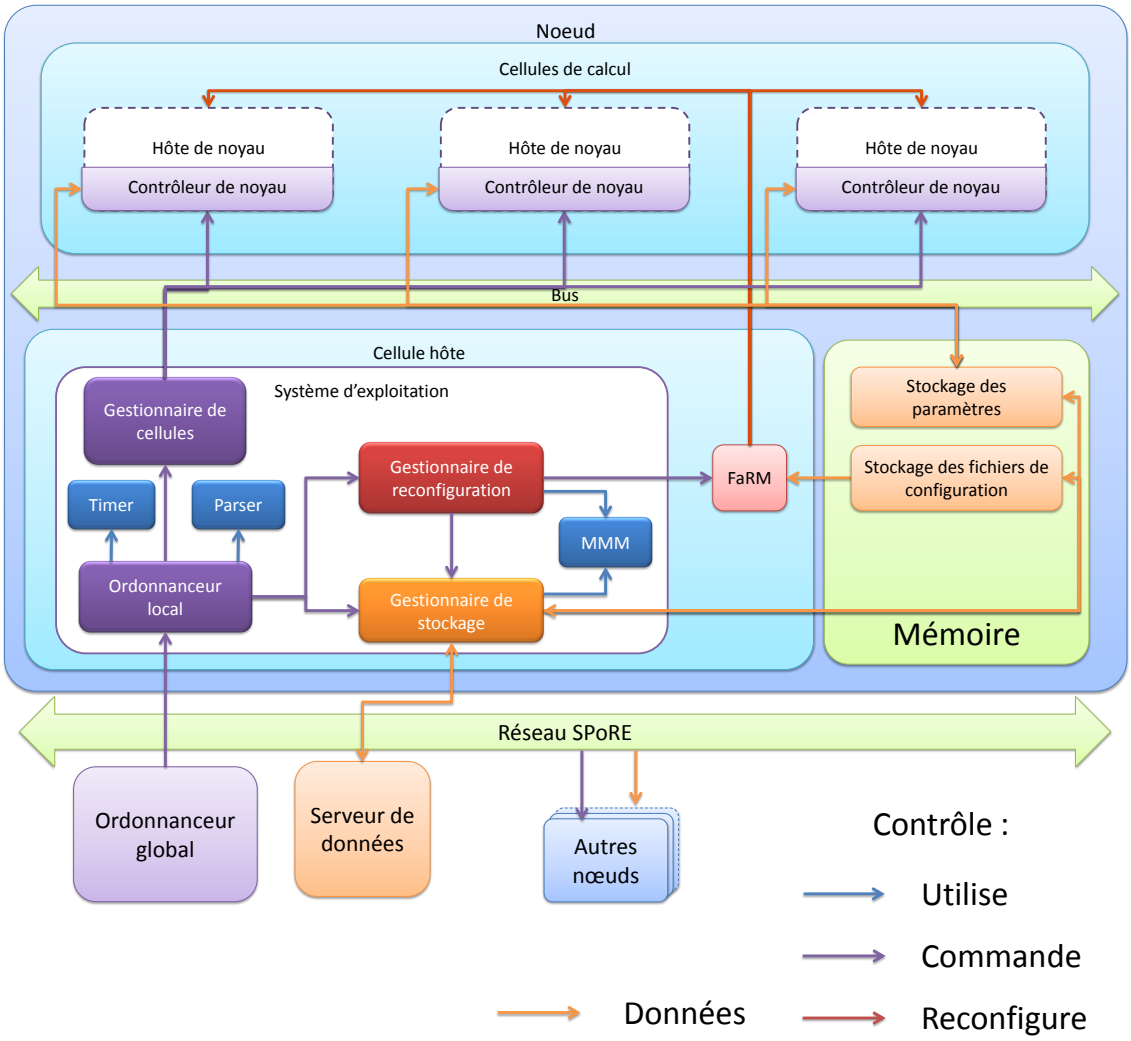


Figure 5.4 – *Organisation des services d'un nœud HSDP.*

global, l'ordonnanceur local reçoit l'adresse d'un serveur de données qu'il devra utiliser par la suite.

Lorsqu'il reçoit un ordre d'exécution de l'ordonnanceur global, il ordonne au gestionnaire de stockage de récupérer le descripteur d'application auprès du serveur de données. Puis, en soumettant le descripteur au parser, il crée une structure de données représentant l'application. Par la suite, à chaque exécution d'un noyau, les données nécessaires sont récupérées sur le serveur de données : descripteurs de noyau, fichiers de configuration, paramètres, etc.

L'ordonnanceur local doit choisir entre les différentes implémentations disponibles pour un noyau, lancer l'exécution de ceux-ci et vérifier s'ils ont fini leur travail. Pour l'instant, l'algorithme d'ordonnancement se contente de choisir prioritairement une implémentation matérielle si elle est disponible et qu'une cellule correspondante est libre. À terme, comme pour l'ordonnanceur global, un ensemble

de paramètres doivent entrer en ligne de compte pour réaliser ce choix. Cette partie sera détaillée dans le chapitre concernant les travaux futurs.

Lite Memory Manager

LMM est une bibliothèque que nous avons développée pour la gestion de la mémoire. Celle-ci est capable de gérer plusieurs portions de mémoire simultanément. Chaque portion mémoire est découpée en blocs d'une taille spécifiée par l'utilisateur. Lorsqu'un élément doit être stocké en mémoire, une requête au LMM indiquant la taille nécessaire retourne une adresse à laquelle il est possible de placer la donnée.

La gestion de la mémoire en blocs consiste à attribuer un nombre de blocs suffisant pour stocker la taille de la donnée concernée. Par exemple, avec des blocs de 10 Kio, pour une donnée de 15 Kio, la librairie attribuera deux blocs. Bien que générant des pertes d'espace mémoire pour chaque bloc non rempli, cette technique permet un fonctionnement rapide de la bibliothèque, et reste adaptée pour l'usage que nous en faisons. En effet, nous utilisons cette bibliothèque pour gérer le stockage des paramètres et des fichiers de configuration afin que les premiers soient accessibles aux noyaux et les seconds à FaRM. Or, ces données se renouvellent mais ne s'accumulent pas : une fois des paramètres consommés, ils sont supprimés de la mémoire. L'utilisation d'une gestion de la mémoire en blocs n'est donc pas un handicap. Pour le stockage des fichiers de configuration sur un nœud sur lequel les hôtes de noyaux sont de taille sensiblement équivalente, il suffira de configurer une taille de bloc légèrement supérieure à celle-ci, chaque fichier occupant alors un bloc.

De nombreuses recherches ont été menées sur la fragmentation mémoire, et notamment sur la pertinence du modèle bloc. On peut citer par exemple les travaux menés par Rezaei et al. [63]. Néanmoins, ce sujet n'étant pas au cœur de notre projet, nous nous contentons de cette simple bibliothèque qui répond à nos besoins.

Gestionnaire de reconfiguration

Le gestionnaire de reconfiguration prend en charge toute la chaîne de la reconfiguration d'un hôte de noyau. Il met en place celle-ci sur la base de l'implémentation et de la cellule choisies par l'ordonnanceur, sur requête de celui-ci.

Le gestionnaire de reconfiguration repose sur un ensemble de trois éléments : le service de reconfiguration, l'IP matériel FaRM et le pilote faisant le lien entre les deux. L'IP FaRM est capable de reconfigurer une zone du FPGA à partir d'un fichier de configuration stocké en mémoire vive. Une zone mémoire est donc réservée au stockage des fichiers de configuration.

Tout d'abord, le service de reconfiguration fait une requête au gestionnaire de stockage pour rapatrier le fichier de configuration si nécessaire. Avec l'aide du LMM,

il détermine une zone de l'espace mémoire disponible pour le stockage du fichier. La taille de la mémoire étant limitée, le gestionnaire utilise une politique d'éviction sur les fichiers les moins récemment utilisés (Least Recently Used – LRU) si nécessaire. Puis, une requête au pilote permet de transférer le contenu du fichier à l'adresse définie, et le pilote attribue une référence à ce fichier.

Ensuite, et tant que le fichier n'est pas évincé au profit d'un autre, il suffit de faire une requête au pilote FaRM pour que celui-ci déclenche l'opération de reconfiguration. Le pilote envoie alors l'adresse et la taille du fichier de configuration à l'IP FaRM, qui gère ensuite automatiquement le transfert des données dans l'ICAP pour reconfigurer le matériel.

L'IP FaRM supporte également plusieurs autres opérations, tel que le préchargement d'un fichier de configuration, qui permet de charger le fichier dans la mémoire locale de l'IP, pour lancer la reconfiguration immédiatement à un instant précis. La relecture de la configuration d'une zone du FPGA en vue de la stocker dans un fichier est également supportée. Ces opérations ont été implémentées dans le pilote, mais ne sont pas utilisée dans cette version de la plateforme.

On pourrait imaginer utiliser la fonction de préchargement lorsque l'on connait à l'avance le prochain noyau devant être ordonnancé, ainsi que la date probable de fin de calcul du noyau précédent. Dans ce cas là, le préchargement permettra de lancer une reconfiguration immédiatement à la fin de l'exécution d'un noyau.

Gestionnaire de stockage

Le gestionnaire de stockage est chargé de la communication avec le serveur de données. C'est à lui que sont adressées les requêtes de téléchargement des descripteurs et autres fichiers, ainsi que d'envoi des résultats sous la forme de fichiers de données vers le serveur. Le gestionnaire de stockage dispose d'un répertoire de stockage (« repository ») en mémoire de masse pour chaque composant des applications : descripteurs de noyaux et d'application, paramètres, etc. Les fichiers téléchargés sur le serveur de données sont stockés dans le répertoire approprié, où ils peuvent ensuite être utilisés par le service qui en a fait la demande.

Les fichiers sont transférés par socket, en ouvrant une connexion entre le serveur et le nœud. Le téléchargement des fichiers est effectué dans un sous-processus logiciel (« thread ») séparé du runtime, afin de permettre au système d'exploitation de traiter indépendamment le téléchargement et l'exécution du runtime. Un fichier à télécharger sera ajouté à une liste de transferts, liste traitée par le thread de téléchargement. Si à un instant donné, un fichier doit être téléchargé immédiatement pour éviter de bloquer l'ordonnancement, alors il peut être marqué comme prioritaire et ramené au sommet de la liste des téléchargements.

Cette méthode de traitement des requêtes par processus indépendant, couplée à la possibilité de traiter un fichier prioritairement si nécessaire, permet de lancer

le téléchargement de tous les fichiers nécessaires pour l'application. Ceux-ci sont donc ajoutés à la liste de téléchargement au fur et à mesure du traitement des descripteurs.

Gestionnaire de cellule

Le service gestionnaire de cellule est destiné à interagir avec le contrôleur de noyau des cellules de calcul. C'est lui qui est chargé de la communication avec le contrôleur de noyau et de l'exécution des accesseurs. Il est ainsi capable de connecter et de déconnecter un hôte de noyau, et de gérer le timer du contrôleur. Concernant les accesseurs, c'est le gestionnaire de cellule qui les prend en charge : il est ainsi capable de réaliser les différents types d'actions définies. C'est donc lui qui applique le décalage nécessaire pour faire correspondre les registres de la cellule aux registres de l'IP présente sur l'hôte de noyau.

Il est également chargé de la gestion de la mémoire des paramètres. Ce segment de la mémoire centrale est réservé aux paramètres des noyaux disposant d'une interface maître. En plaçant les données dans cette mémoire, les différents IPs matériels peuvent alors venir les chercher, puis y placer les résultats en vue de leur consommation par un autre IP. Le gestionnaire de cellule peut enfin récupérer les résultats du dernier noyau à la fin du traitement.

En manipulant les accesseurs, le gestionnaire de cellule joue un rôle central dans la couche de virtualisation de SPoRE. En effet, les accesseurs sont utilisés pour toutes les interactions avec les cellules : configuration initiale, lancement de l'exécution, vérification de la fin d'exécution, etc. En se plaçant entre les cellules et les autres services, le gestionnaire de cellules est donc le passage obligé pour toutes les interactions avec celles-ci.

Service de mesure de durées

Le service de mesure de durée est destiné à offrir les mêmes services que le timer matériel pouvant être intégré dans les cellules de calcul, mais pour les processus logiciels. Il permet de créer, manipuler et détruire différents timers. Il est ainsi possible de créer un timer pour chaque processus logiciel, option facultative qui peut être activée dans l'ordonnanceur. Ce service utilise les procédures de mesure de temps de Linux, et offre donc une précision à la microseconde. Dans ce runtime, ce service est utilisé pour mesurer les durées d'exécution des noyaux logiciels.

Parser XML

Le parser XML est destiné à récupérer les informations contenues dans les descripteurs au format XML pour placer celles-ci dans des structures de données exploitables par les autres services. Lorsqu'un descripteur lui est transmis pour en

extraire les données, le parser commande automatiquement le téléchargement des autres descripteurs mentionnés à l'intérieur, en prévision de leur usage futur. Par exemple, lors du traitement d'un descripteur d'application, le parser demandera automatiquement le téléchargement des différents descripteurs de noyaux qui y sont mentionnés. La syntaxe des descripteurs XML est précisée dans la section suivante.

5.1.2.d Bilan logiciel

Avec plus de 10 000 lignes de code en C pour 250 Kio d'exécutable final, le runtime de la cellule hôte est un ensemble logiciel complexe. Bien que la séparation en services permette d'isoler les routines selon leur fonctionnalité, l'ensemble reste imposant à première vue. Néanmoins, du point de vue du développeur d'applications, cet ensemble permet une automatisation totale du processus de reconfiguration partielle : son seul besoin est de générer le fichier de configuration partiel correspondant à son code, et tout est ensuite pris en charge. Concernant l'interaction avec l'IP, un effort supplémentaire est requis pour écrire les différents descripteurs, dont nous allons expliciter la syntaxe dans la prochaine section.

5.1.3 Limitations de la plateforme HSDP par rapport à SPoRE

Comme indiqué, la principale limitation de la plateforme concerne l'ordonnancement. En effet, les métriques permettant le choix dans la répartition des noyaux n'ont pas été implémentées. Il n'est donc pas possible à l'heure actuelle, de répartir l'exécution des noyaux d'une application sur plusieurs nœuds, faute de critères pour le faire. En revanche, il est possible d'exécuter plusieurs applications simultanément, chacune sur un nœud. Par ailleurs, l'ordonnanceur local actuellement implémenté dans la plateforme ne permet pas les opérations de contrôle du modèle, telles que les boucles et les tests conditionnels. Néanmoins, les services du runtime étant modulaires, ceci n'est pas réellement une limitation liée à la plateforme, mais correspond simplement à une implémentation manquante.

La seconde limitation concerne l'utilisation de MPI. Celui-ci ayant été implémenté avec succès sur la première plateforme, nous n'avons pas jugé nécessaire de l'intégrer ici, d'autant plus que les IPs matériels n'y font pas appel. En revanche, celui-ci pourrait être utile pour l'exécution des noyaux logiciels exécutés sur la cellule hôte. L'utilisation de réseaux MPI matériels pourrait être envisagée pour ajouter la compatibilité avec les IPs matériels supportant ce protocole. Il existe déjà des implémentations matérielles de MPI, telles celles présentées en [48, 35, 92]. Par ailleurs, l'utilisation d'un réseau de communication reconfigurable, sur le modèle de ReMAP [82], permettrait plusieurs avantages ; tout d'abord, quant au nombre d'utilisation des ressources par le réseau. En effet, si les liens entre les unités d'exécution sont créés dynamiquement, on autorise alors un certain nombre de communications

simultanées. Or, ce nombre est forcément inférieur au pire cas, dans lequel tous les PEs communiquent simultanément. En effet, si l'on envisage ce cas, on aurait alors à faire à un réseau de type « crossbar », et l'aspect dynamique serait alors inutile. Un second avantage concerne les opérations de réduction, communes dans les communications MPI. Celles-ci pouvant être réalisées matériellement au cours de la communication, on déleste les unités d'exécution de ces mêmes calculs.

5.2 Syntaxe et structure des descripteurs

Afin de décrire une application pour HSDP, on utilise un ensemble de descripteurs, présentés sur la figure 5.5. Au plus haut niveau, le *descripteur d'application*

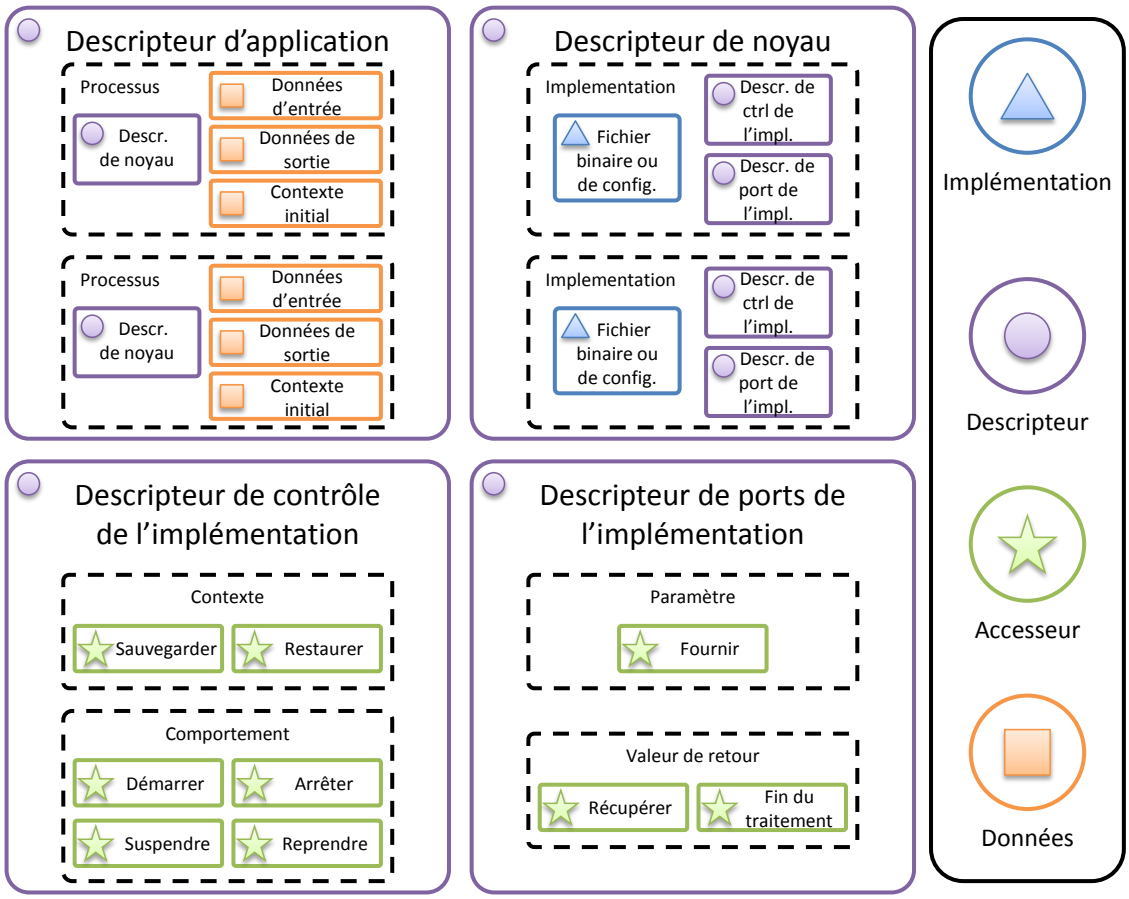


Figure 5.5 – *Hiérarchie des descripteurs dans une application SPoRE.*

répertorie les différents noyaux et leurs relations. C'est ce fichier qui contient la partie contrôle de l'application. Le descripteur d'application fait donc référence à des *descripteurs de noyaux*, chacun chargé de représenter une opération à réaliser sur des données. C'est au sein du descripteur de noyau que sont référencées les différentes implémentations disponibles pour celui-ci, chacune mettant à disposition le binaire logiciel ou le fichier de configuration matériel correspondant. Enfin,

chaque implémentation fait référence à deux fichiers contenant les accesseurs nécessaires pour la virtualisation : le *descripteur de contrôle de l'implémentation* et le *descripteur de ports de l'implémentation.*

Tous ces descripteurs utilisent une syntaxe XML pour permettre l'indépendance vis-à-vis de la plateforme et donc la portabilité. Dans cette section, nous allons nous atteler à décrire chacun de ces fichiers, leur structure et leur syntaxe. Nous évoquerons également les fichiers de données, qui permettent le stockage de certaines informations nécessaires à la configuration des noyaux.

5.2.1 Structure commune

La base commune à tous les descripteurs comprend une racine, nommée en fonction de la catégorie du descripteur, et permettant au parser d'identifier le contenu de celui-ci. Ainsi, en cas d'erreur sur le type de descripteur, l'utilisateur est immédiatement informé sans que le runtime SPoRE tente d'interpréter celui-ci.

La racine est dotée d'un attribut, nommé ID, qui est une référence numérique unique sur l'ensemble des descripteurs présents dans l'environnement SPoRE. Afin d'éviter les confusions, il n'est pas possible pour deux fichiers présents dans le système, même appartenant à deux applications différentes, d'utiliser le même ID. Ainsi, un descripteur de noyau possédant le même ID dans deux applications différentes est bien la représentation du même noyau. Dans le contenu des descripteurs, il est possible de définir un nom de fichier lorsque l'on fait référence à un descripteur, en utilisant l'attribut *File_ name*, afin de permettre à l'utilisateur d'identifier plus facilement les fichiers présents dans le système. Néanmoins, le seul identifiant réellement utilisé par le système est l'ID, et le nom de fichier est une simple recommandation pour le système, qui peut éventuellement ne pas en tenir compte. S'il n'est pas possible d'utiliser le nom de fichier défini, le runtime utilisera une dénomination standard de type 0xABCDEFGH, avec ABCDEFGH représentant l'ID du fichier en hexadécimal sur 8 caractères.

5.2.2 Le descripteur d'application

Le descripteur d'application représente le plus haut niveau de description d'une application SPoRE. Celui-ci contient la partie contrôle qui décrit les relations entre les noyaux à exécuter. Afin d'exécuter une application, l'utilisateur fournit la référence de ce fichier à l'ordonnanceur global. L'ordonnanceur global peut, à partir de celui-ci créer plusieurs descripteurs d'applications représentant des sous-applications, qu'il soumet ensuite aux ordonnanceurs locaux des nœuds. Le descripteur d'application est totalement portable d'une plateforme à une autre, car il fait référence aux noyaux virtuels, sans considération pour l'implémentation réelle. La structure XML de celui-ci est présentée sur la figure 5.6.

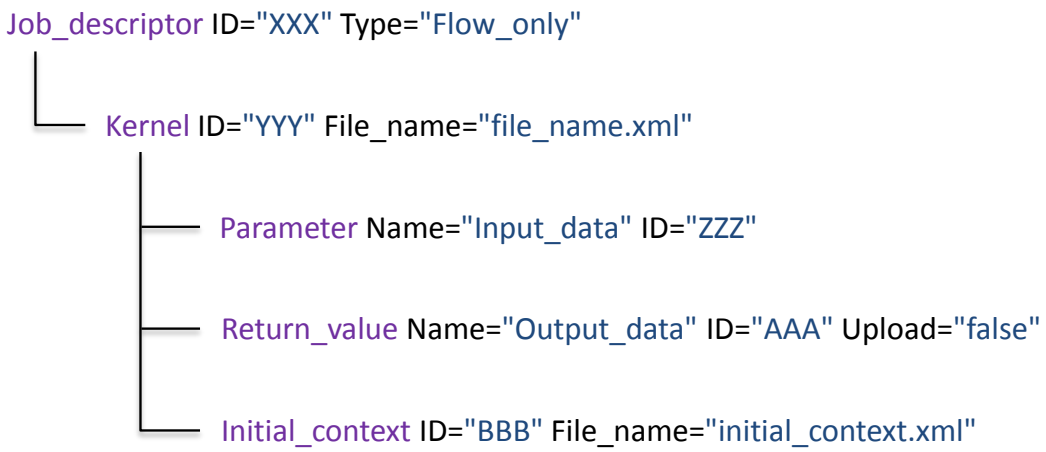


Figure 5.6 – *Structure des descripteurs d'application.*

Le descripteur d'application contient un ensemble de sous-éléments de type *Kernel*, représentant chacun un noyau à exécuter. Comme indiqué précédemment, le nom de fichier éventuellement proposé pour le noyau est une indication que le système utilisera si possible.

Pour chaque noyau, on indique une référence pour les paramètres et les valeurs de retour. On trouvera donc une clé *Parameter* pour chaque paramètre d'entrée du noyau, et de même pour les valeurs de retour (paramètres de sortie) avec *Return_value*. Chacun de ceux-ci contient deux attributs : *Name* et *ID*. L'attribut *Name* doit correspondre au nom définit par le descripteur de noyau pour ses entrées et sorties. Ainsi, si un descripteur de noyau fournit deux paramètres *Parametre_1* et *Parametre_2* et une valeur de retour *Parametre_3*, le descripteur d'application disposera de trois lignes dans la référence au noyau, indiquant ces noms. Ceci permet d'associer chaque paramètre avec une référence (identifiant – ID) de donnée unique. Il est à noter que tous les paramètres définis par un descripteur de noyau ne doivent pas forcément être utilisés par un descripteur d'application, si ceux-ci ne sont pas nécessaires dans ce cas précis.

L'identifiant *ID* correspondant à un fichier de données est utilisé par l'ordonnanceur pour faire le lien entre les noyaux. Ainsi, si tous les fichiers de données nécessaires en entrée d'un noyau existent, celui-ci pourra être ordonnancé. En revanche, tant qu'il manque au moins un fichier, c'est que l'exécution d'un noyau précédent est encore nécessaire. L'attribut *upload* permet d'indiquer au runtime local si le fichier doit être envoyé vers le serveur de données, ou s'il ne s'agit que d'un résultat intermédiaire qui ne doit pas être récupéré.

Les boucles et les tests ont été uniquement définis à l'état théorique, et n'ont pas été testés. En effet, cette partie est liée à l'ordonnancement, qui est la partie restant à réaliser sur cette plateforme. Nous ne présenterons donc pas ceux-ci en détail ici, car ils n'ont pas encore été validés. Néanmoins, la structure que nous

avons choisie pour le modèle d'application rend la syntaxe de base relativement simple : il suffit en effet de définir des clés de type *For*, *While* ou encore *If*.

5.2.3 Le descripteur de noyau

Le descripteur de noyau comprend une ou plusieurs balises *Implementation*, énumérant les différentes implémentations disponibles. On présente sa structure sur la figure 5.7.

Chaque implémentation doit disposer d'une balise *Bitstream*, *Binary* ou *Command*, correspondant à une implémentation matérielle ou logicielle, et en indiquant le fichier principal. La figure 5.7 présente un noyau disposant de deux implémentations. L'une, matérielle, fait référence à un fichier de configuration (« Bitstream »). L'autre, logicielle, utilise une commande pour lancer un exécutable local. En plus de ce fichier principal, on indique les références au descripteur de contrôle de l'implémentation et au descripteur de port de l'implémentation. Le descripteur de noyau fait donc le lien entre l'application elle-même et l'implémentation particulière qui en est faite, en indiquant les fichiers contenant les accesseurs, qui seront utilisés par la couche de virtualisation.

5.2.4 Les descripteurs contenant des accesseurs

Les deux autres descripteurs sont destinés aux accesseurs. Pour chaque type d'opération sur le noyau, un accesseur présente les actions successives à réaliser pour réaliser l'action escomptée. Les accesseurs disposent de deux variantes selon que l'implémentation est logicielle ou matérielle. On présente dans un premier temps la structure des deux descripteurs, pour ensuite nous intéresser à la syntaxe des accesseurs.

5.2.4.a Le descripteur de contrôle de l'implémentation

Le descripteur de contrôle de l'implémentation est destiné à indiquer comment interagir avec l'implémentation pour les opérations concernant le comportement et le contexte. Par comportement, on entend les opérations affectant l'état du noyau : démarrage, suspension, reprise et arrêt. Les opérations sur le contexte consistent en la sauvegarde de celui-ci, en vue par exemple d'une migration, et sa restauration, pour permettre la reprise du traitement dans un état précis mais également le chargement du contexte initial. La structure de ce descripteur est présentée sur la figure 5.8.

La présence de tous les accesseurs n'est pas nécessaire, certains pouvant être omis. Dans certains cas, il est même possible que le descripteur de contrôle de l'implémentation soit inexistant, si l'IP n'a pas besoin d'action particulière pour lancer son traitement.

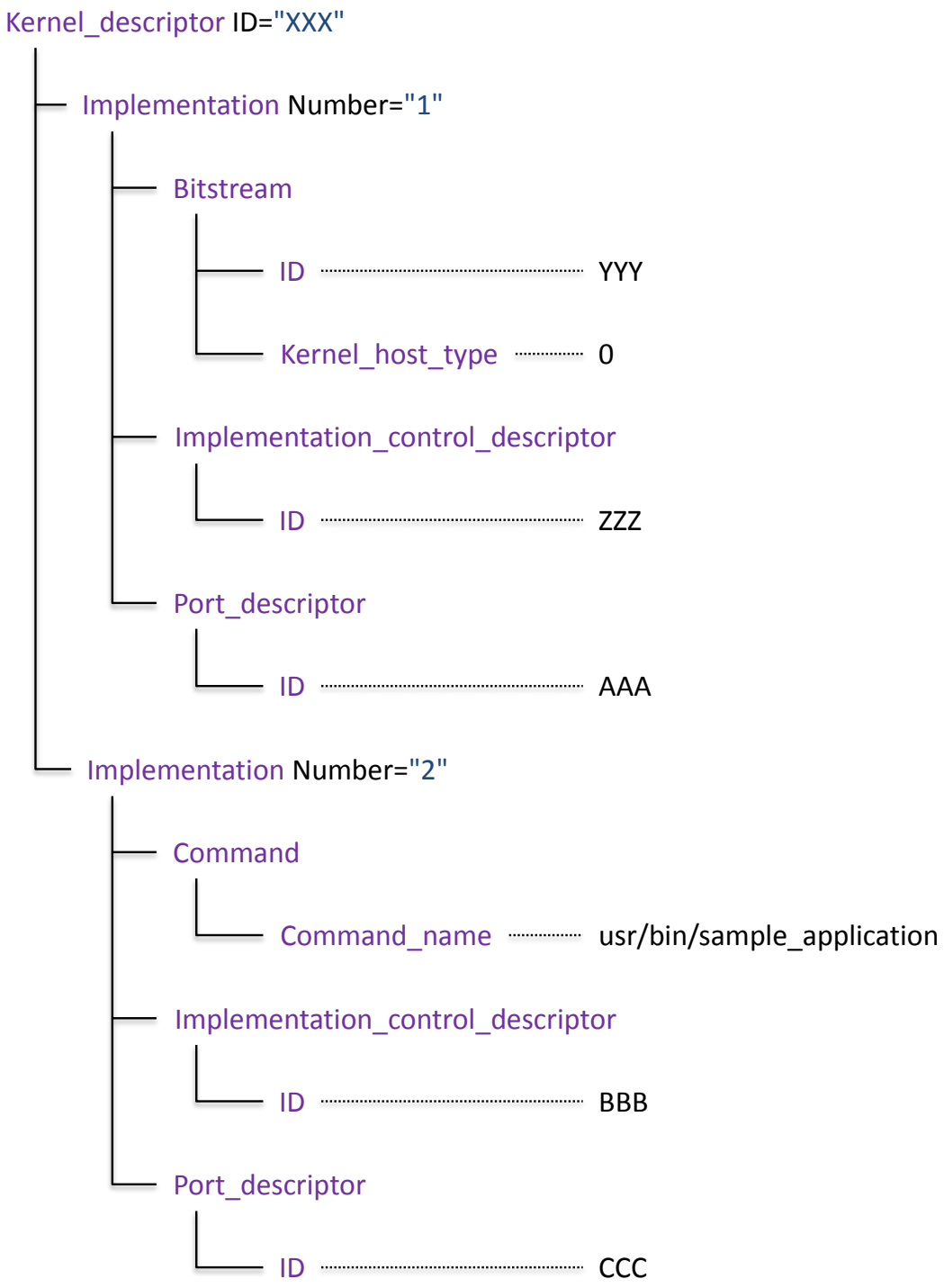


Figure 5.7 – *Structure des descripteurs de noyau.*

5.2.4.b Le descripteur de ports de l'implémentation

Le descripteur de ports de l'implémentation sert à rassembler les informations sur les échanges du noyau avec son environnement. Ainsi, c'est lui qui contient les accesseurs concernant les paramètres et valeurs de retour. Sur une plateforme sup-

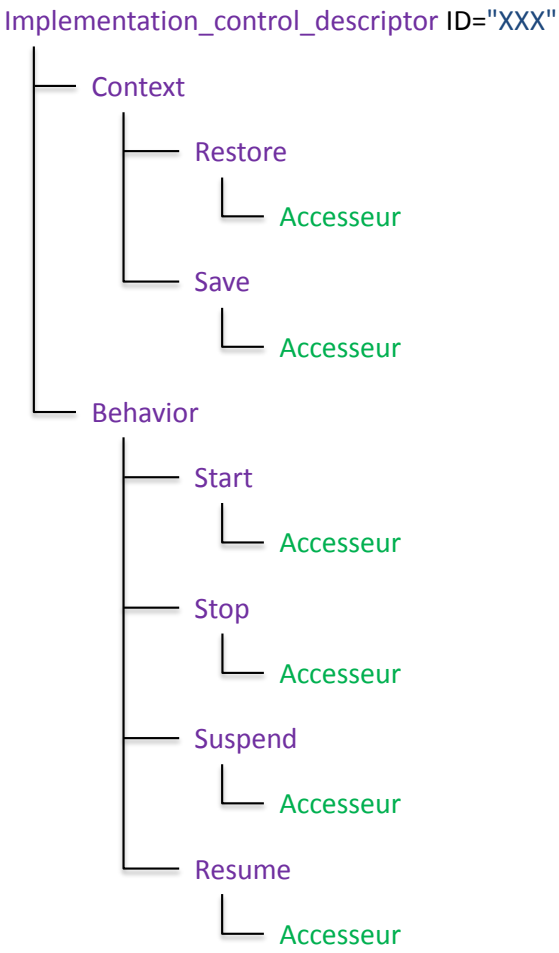


Figure 5.8 – *Structure des descripteurs de contrôle de l'implémentation.*

portant la communication directe entre noyaux, il intègrerait également la description des opérations de communication. Sa structure est présentée sur la figure 5.9.

On distingue deux types de clés, *Parameter* et *Return_value*, correspondant respectivement aux paramètres d'entrée et de sortie du noyau. Ceux-ci, en nombre variable, disposent d'un nom, d'une taille et d'une quantité. Le nom est celui qui sera utilisé dans le descripteur d'application pour y associer un fichier de données. La taille et la quantité, en octets, indiquent respectivement la taille et le nombre de mots concernés, donnant la quantité de données transitant par ce port. Enfin, les accesseurs indiquent comment fournir les paramètres au noyau (« Get »), comment récupérer les valeurs de retour (« Set »), mais également comment savoir si une valeur de retour est disponible (« Return_value_ready_hint »).

5.2.4.c Syntaxe des accesseurs

Le cœur de ces deux descripteurs est formé d'accesseurs. Les accesseurs sont des séries d'action à effectuer sur un noyau pour réaliser une opération particu-

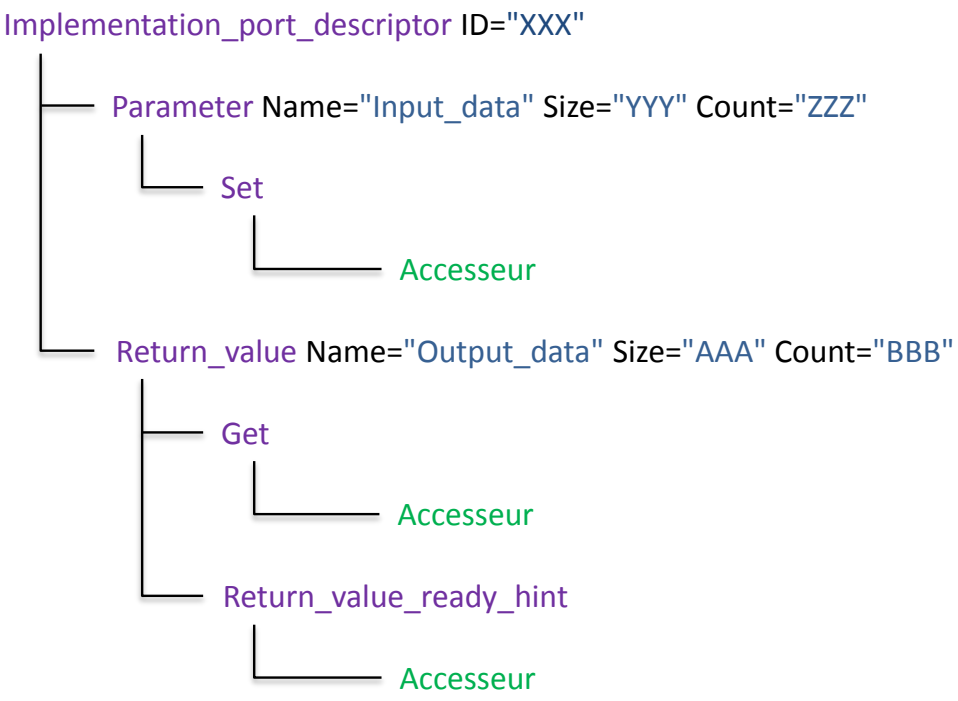


Figure 5.9 – *Structure des descripteurs de ports de l'implémentation.*

lière. Selon la nature du noyau, logiciel ou matériel, la syntaxe des accesseurs est différente.

Accesseur matériel

Les actions des accesseurs matériels opèrent sur des registres ou des plages mémoire. Un accesseur matériel est constitué d'une liste de clés *Action*, comme présenté sur la figure 5.10 [1]. Chaque action contient plusieurs attributs standards : *Type*, *Direction*, *Offset*, *Size* et *Count*.

L'attribut *Offset* indique l'adresse du registre ou de la première adresse mémoire concernée, par rapport à l'adresse de base de l'IP sur le bus. Les attributs *Count* et *Size* indiquent respectivement le nombre de mots concernés et la taille de ceux-ci. L'attribut *Direction* précise s'il s'agit d'une lecture ou d'une écriture.

Le type indique si l'action se réfère à un registre, une plage mémoire ou bien si l'on opère sur un pointeur en mémoire vive. Le type FIFO indique plusieurs actions successives sur un même registre. Dans le cas du pointeur, une ou plusieurs sous-clés sont présentes : *Address*, *Size* et *Count*. La clé *Address* indique dans quel registre de l'IP l'adresse mémoire doit être écrite. Cette clé doit toujours être présente. Si nécessaire, les clés *Count* et *Size* indiquent à quel registre de l'IP le nombre

1. On précise que le contenu d'une clé XML se situe entre le caractère "<" et les caractères "/>". Le retour à la ligne présent sur cette figure n'est du qu'à des limitations de largeur de la mise en page.

```
<Action Type="Memory" Direction="Write"
    Data_reference="0" Offset="36" Size="4" Count="4"/>
<Action Type="Register" Direction="Write"
    Data_constant="1" Offset="0" Size="4"/>
<Action Type="FIFO" Direction="Write"
    Data_reference="16" Offset="40" Size="4" Count="4"/>
<Action Type="Pointer" Direction="Read"
    Data_parameter="" Count="0x00028000" Size="16">
      <Address Offset="28" Size="4"/>
      <Size Offset="32" Size="4"/>
      <Count Offset="36" Size="4"/>
</Action>
```

Figure 5.10 – *Exemple d'actions possibles dans un accesseur matériel.*

d'éléments et leur taille doivent être transmis en mémoire.

Enfin, un autre attribut est présent, qui peut prendre plusieurs noms : *Data_reference*, *Data_constant*, *Data_parameter* ou *Data_ignored*. Si l'attribut *Data_constant* est présent, alors c'est la valeur associée à cet attribut qui doit être écrite. Dans le cas de *Data_reference*, la valeur de l'attribut est l'index d'une valeur dans un fichier de données. Dans ce cas, s'il s'agit d'une écriture, le runtime va chercher la valeur dans le fichier de données lié à l'accesseur : celui précisé dans le descripteur pour le contexte par exemple. À l'inverse, dans le cas d'une lecture, la valeur lue sera stockée à cet index dans un ficher de données. Ceci permet d'utiliser dans un accesseur une valeur variant selon le programme, voire même une valeur crée par un noyau précédent.

L'utilisation de *Data_ignored* n'est utile qu'en lecture : cela signifie que la valeur lue sera tout simplement ignorée par le runtime. En effet, dans certains cas, il faut venir lire un registre pour débloquer une action dans un IP. Par exemple, pour réinitialiser un drapeau, il est parfois nécessaire de lire un registre, mais la valeur lue est déjà connue à l'avance et peut donc être ignorée.

Enfin, l'attribut *Data_parameter* est un cas particulier, réservé aux accesseurs Get et Set des descripteurs de ports. Cela signifie que, dans un accesseur comprenant plusieurs actions, c'est cette action là qui réalise l'accès au paramètre. Par exemple, si pour écrire un paramètre, il faut d'abord configurer un registre avec une valeur, on aura un accesseur avec deux actions : une *Data_constant* ou bien une *Data_reference*, puis une *Data_parameter*. Dans le cas d'une écriture, le contenu de cette deuxième action sera celui du fichier de données correspondant au paramètre. Dans le cas d'une lecture, cette deuxième action fournira le contenu à stocker dans le fichier de données correspondant au paramètre.

Accesseur logiciel

Les accesseurs logiciels sont bien plus simples que les accesseurs matériels. En effet, dans notre cas, les noyaux logiciels sont des programmes fonctionnant sur la cellule hôte. Il s'agit donc, pour interagir avec ceux-ci, d'instruction en ligne de commande. La figure 5.11 présente des actions possibles pour un accesseur logiciel.

```
<Action Type="String" Prefix="param2" Data_ignored=""/>
<Action Type="String" Prefix="param1" Data_ignored=""
  Offset="1"/>
<Action Type="String" Data_reference="4" Offset="2"/>
<Action Type="String" Prefix="file" Offset="3"/>
<Action Type="File" Data_parameter="" Offset="4"/>
```

Figure 5.11 – *Exemple d'actions possibles dans un accesseur logiciel.*

Les paramètres UNIX sont des chaînes de caractères séparés par une espace. Certains paramètres se présentent par paire, sous la forme de deux paramètres séparés ou concaténés. Par exemple, lors de la compilation d'une application avec GNU C Compiler (GCC), on utilise la ligne de commande suivante :

```
gcc <fichier.c> -O2 -I<dossier> -o <fichier.elf>
```

Dans cette commande, le paramètre -02, indiquant un niveau d'optimisation, se présente seul. En revanche, le paramètre -I, permettant d'ajouter un dossier contenant des fichiers d'en-tête, est concaténé à sa valeur. Enfin, le paramètre -o, servant à indiquer le nom souhaité pour le fichier exécutable, se présente séparé de sa valeur. Ainsi, -o et le nom du fichier représentent deux paramètres distincts car séparés par un espace.

Pour gérer ces différents cas de figure, nos actions sont composées de deux éléments : un préfixe et une donnée. Si le paramètre est unique et contient un seul élément (-O2), seul le préfixe est présent, et la donnée est alors *Data_ ignored*. Si le paramètre est unique mais contient deux éléments (-I<dossier>), l'action contiendra le préfixe ainsi que la donnée. Enfin, si le paramètre est composé de deux chaînes qui se suivent (-o <fichier.elf>), on aura une action avec le seul préfixe, et une action avec la donnée.

L'indication de l'offset, qui correspond à la position du paramètre dans la liste, permet de s'assurer que les deux paramètres se suivront, ou bien de respecter une contrainte de position. Si l'offset n'est pas indiqué, le runtime placera le paramètre à une position non utilisée par un autre paramètre. Enfin, le type *File* permet d'indiquer que le contenu du paramètre doit être le chemin du fichier de données.

5.2.5 Les fichiers de données

Les fichiers de données ne sont pas des descripteurs à proprement parler, mais certains utilisent une syntaxe similaire. En effet, SPoRE supporte deux types de fichiers de données : les données brutes et les données ordonnées. Les données brutes se présentent sous la forme d'un fichier binaire ne comportant que les données elles-mêmes. La taille du fichier est donc strictement égale à la quantité de données, ce qui évite une perte de place.

En revanche, les données ordonnées permettent d'ajouter des méta-informations sur les données : index, taille, type, comme présenté sur la figure 5.12. Ces fichiers

```
<Data ID="XXX">
  <Value Index="0" Size="4" Content="0x01234567" />
  <Value Index="1" Size="4" Content="0x89ABCDEF" />
  <Value Index="3" Type="String" Content="AbCdEfGhIjKlMnOp" />
</Data>
```

Figure 5.12 – *Exemple de fichier de données ordonnées.*

de données sont notamment utilisés pour contenir des données de configuration, auxquelles il sera possible de se référer en utilisant l'attribut *Data_ reference*.

5.3 Test et validation de l'implémentation

Tout comme la première plateforme, celle-ci devra être testée au moyen d'une application. L'objectif de cette seconde implémentation de SPoRE est différent de celui de la précédente, aussi l'application de test devra être choisie selon cet objectif.

5.3.1 L'application de test

Pour la première plateforme, nous avions choisi une application testant principalement les communications entre les PEs, pour éprouver l'architecture MPI. Pour celle-ci, nous avons besoin d'une application de type flot de données, c'est-à-dire pour laquelle les communications sont de type dépendances de données. De plus, cette application devra comporter des noyaux matériels pour permettre de tester la reconfiguration matérielle. L'existence de ces mêmes noyaux selon une implémentation logicielle permettra également de démontrer la prise en charge des deux types de noyaux.

Pour cette application, l'idée était de réutiliser des IPs matériels statiques existants afin de démontrer la facilité d'intégration de ceux-ci dans notre structure d'application. Nous avons choisi de réaliser un flot de cryptographie : cryptage puis décryptage d'un message à l'aide de l'algorithme Advanced Encryption Standard

(AES). En effet, des implémentations matérielles et logicielles sous licence libre existent pour cet algorithme, ce qui nous permettra de les réutiliser.

La structure est de type flot de données, comportant deux noyaux dont les données de sortie du premier sont réutilisées en entrée par le second, comme présenté sur la figure 5.13. À la fin de la chaîne, les données de sortie doivent être les mêmes

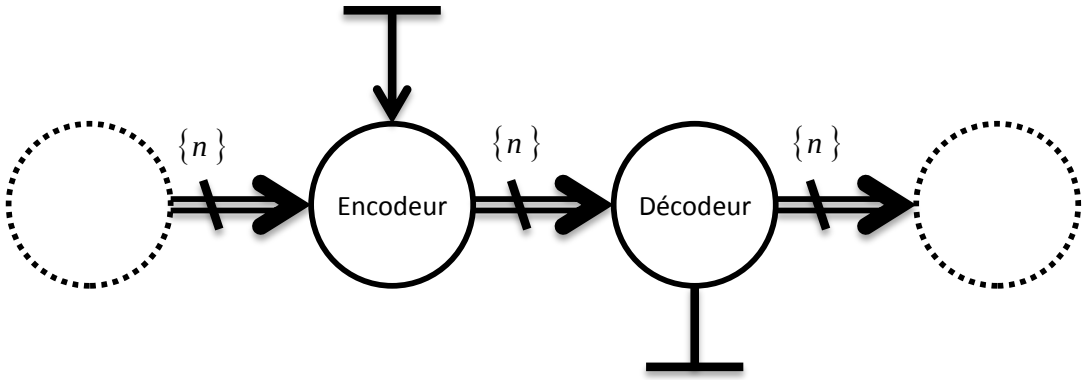


Figure 5.13 – *Modélisation de l'application de test.*

que celles d'entrée. On présente le descripteur d'application sur la figure 5.14. L'ID

```
<Job_descriptor ID="0" Type="Flow_only">
  <Kernel ID="9" File_name="encoder_kernel_descriptor.xml">
    <Parameter Name="Input_data" ID="11"/>
    <Return_value Name="Output_data" ID="12" Upload="false"/>
    <Initial_context ID="5"
      File_name="encoder_0_initial_context.xml"/>
  </Kernel>
  <Kernel ID="10" File_name="decoder_kernel_descriptor.xml">
    <Parameter Name="Input_data" ID="12"/>
    <Return_value Name="Output_data" ID="13" Upload="true"/>
    <Initial_context ID="6"
      File_name="decoder_0_initial_context.xml"/>
  </Kernel>
</Job_descriptor>
```

Figure 5.14 – *Descripteur d'application de notre application de test.*

du paramètre de sortie du premier noyau étant le même que celui du paramètre d'entrée du noyau suivant, le runtime fait automatiquement le lien et sait que le deuxième noyau ne devra être ordonnancé que lorsque le premier aura fini son traitement.

Cette application nécessite deux noyaux, afin de crypter et de décrypter les données. Nous souhaitons disposer, pour chacun de ces noyaux, d'une implémen-

tation logicielle et d'une implémentation matérielle. On utilise donc le descripteur de noyau de la figure 5.15 pour indiquer les deux implémentations disponibles pour l'encodeur. Celui du décodeur sera similaire.

```
<Kernel_descriptor ID="9" Name="AES encoder">
  <Implementation Number="1" Name="Hardware AES encoder">
    <Bitstream>
      <ID>1</ID>
      <Kernel_host_type>0</Kernel_host_type>
    </Bitstream>
    <Implementation_control_descriptor>
      <ID>19</ID>
    </Implementation_control_descriptor>
    <Implementation_port_descriptor>
      <ID>17</ID>
    </implementation_port_descriptor>
  </Implementation>
  <Implementation Number="2" Name="Software AES encoder">
    <Command>
      <Command_name>/usr/bin/openssl</Command_name>
    </Command>
    <Implementation_control_descriptor>
      <ID>20</ID>
    </Implementation_control_descriptor>
    <Implementation_port_descriptor>
      <ID>18</ID>
    </Implementation_port_descriptor>
  </Implementation>
</Kernel_descriptor>
```

Figure 5.15 – *Descripteur de noyau présentant les deux implémentations de l'encodeur.*

5.3.1.a Implémentation des noyaux logiciels

Concernant l'implémentation logicielle, qui s'exécutera sur la cellule hôte, nous pouvons soit distribuer l'exécutable binaire, soit faire appel à un exécutable déjà existant dans le système de fichiers. Or, nos plateformes disposent déjà de la bibliothèque OpenSSL, qui est utilisée notamment pour sécuriser les communications transitant par OpenSSH. En plus de la bibliothèque, pouvant être utilisée par des applications, OpenSSL propose un programme en ligne de commande permettant la réalisation d'opérations cryptographiques, notamment au travers de l'algorithme AES. Le choix d'OpenSSL s'est donc imposé à nous comme le moyen le plus simple d'implémenter les noyaux AES logiciels, permettant d'illustrer la réutilisation.

Afin d'utiliser l'algorithme AES, il faut fournir à OpenSSL les paramètres de configuration correspondant. Pour cela, on fournit le descripteur de contrôle présenté sur la figure 5.16. Les deux dernières actions indiquent que les paramètres 2

```
<Implementation_control_descriptor ID="20">
  <Context>
    <Restore>
      <Action Type="String" Prefix="enc"
        Data_ignored="" Offset="1"/>
      <Action Type="String" Prefix="-e"
        Data_ignored=""/>
      <Action Type="String" Prefix="-aes-128-ecb"
        Data_ignored=""/>
      <Action Type="String" Prefix="-nosalt"
        Data_ignored=""/>
      <Action Type="String" Prefix="-nopad"
        Data_ignored=""/>
      <Action Type="String" Prefix="-K" Offset="2"
        Data_ignored=""/>
      <Action Type="String" Data_reference="4"
        Offset="3"/>
    </Restore>
  </Context>
</Implementation_control_descriptor>
```

Figure 5.16 – *Descripteur de contrôle de l'implémentation logicielle du noyau encodeur AES.*

et 3 sont respectivement "-K" et le contenu du fichier de données. Ceci permet de configurer la clé de cryptage.

Afin de configurer les fichiers de données d'entrée et de sortie, on utilise le descripteur de ports de la figure 5.17. Celui-ci indique au runtime que les fichiers de données d'entrée et de sortie devront être respectivement précédés du paramètre "-in" et "-out".

5.3.1.b Implémentation des noyaux matériels

Pour l'implémentation matérielle, on utilise un IP AES disponible sur OpenCores [60]. Celui-ci est un IP purement flot de données qui n'est pas prévu pour être connecté sur un bus. En effet, il dispose de différents signaux permettant de fournir la clé de cryptage, les données ou encore de démarrer les calculs. Afin de l'utiliser sur un bus, ces différents signaux doivent être mappés sur des registres mémoire. Enfin, nous souhaitons octroyer à ce noyau une interface maître, afin d'autoriser

```
<Implementation_port_descriptor ID="18">
  <Parameter Name="Input_data">
    <Set>
      <!--Configure input file-->
      <Action Type="String" Prefix="-in"
        Offset="8" Data_ignored=""/>
      <Action Type="File" Data_parameter=""
        Offset="9"/>
    </Set>
  </Parameter>
  <Return_value Name="Output_data">
    <Get>
      <!--Configure output file-->
      <Action Type="String" Prefix="-out" Offset="10"
        Data_ignored=""/>
      <Action Type="File" Data_parameter=""
        Offset="11"/>
    </Get>
  </Return_value>
</Implementation_port_descriptor>
```

Figure 5.17 – *Descripteur de port de l'implémentation logicielle de l'application AES.*

une gestion par pointeurs.

De plus, cet IP offrant à la fois un algorithme de cryptage et de décryptage, nous souhaitons séparer ces deux fonctionnalités. Un signal sur un bit est utilisé pour choisir la fonctionnalité utilisée. En fournissant une constante sur la valeur de ce signal, on rend celui-ci statique. Ainsi, lors de l'opération de synthèse, les éléments correspondant à la fonction qui n'est pas choisie seront supprimés par simplification par l'outil, créant donc deux IPs distincts.

Pour la création de l'interface bus de l'IP, celle-ci doit être au format IPIC pour pouvoir se connecter sur les ports de l'hôte de noyau. Le format IPIC est utilisé par Xilinx pour tous ses IPs : ceci permet, quel que soit le bus, d'utiliser une connexion commune en générant automatiquement une interface faisant le lien entre l'IPIC et le bus.

La création de l'interface esclave de l'IP est quasi-immédiate : il suffit de venir écrire ou lire chaque signal de l'IP selon le registre sélectionné. En revanche, la partie maître, facultative, nécessite un peu plus de travail, car il faut écrire une machine à état, qui ira lire les données dans la mémoire pour les fournir à l'IP. Néanmoins, à terme, ce DMA pourra être intégré directement dans le contrôleur de noyau pour permettre l'utilisation de l'interface maître par des IPs qui en sont dépourvues. Nous obtenons finalement les tailles indiquées dans le tableau 5.2.

Resource	Encodeur	Décodeur
LUTs	2 430 (5,4%)	2 975 (6,6%)
BRAMs	0 (0,0%)	0 (0,0%)
DSP48Es	0 (0,0%)	0 (0,0%)

Tableau 5.2 – *Ressources utilisées par les noyaux AES. Les pourcentages indiquent l'occupation du composant Virtex 5 fx70t.*

Pour l'implémentation matérielle, on utilise le descripteur de contrôle de l'implémentation de la figure 5.18. Celui-ci ne contient donc qu'une ligne pour indiquer

```
<Implementation_control_descriptor ID="19">
  <Context>
    <Restore>
      <Action Type="Memory" Direction="Write"
        Data_reference="0" Offset="36" Size="4" Count="4"/>
    </Restore>
  </Context>
  <Behavior>
    <Start>
      <Action Type="Register" Direction="Write"
        Data_constant="01" Offset="20" Size="4"/>
    </Start>
  </Behavior>
</Implementation_control_descriptor>
```

Figure 5.18 – *Descripteur de contrôle de l'implémentation logicielle du noyau encodeur AES.*

comment fournir la clé, et une ligne pour la méthode permettant de démarrer les calculs. Le moyen de fournir et de récupérer les données est, lui, pris en charge par le descripteur de ports de la figure 5.19. Les données sont ici passées par pointeurs, c'est-à-dire que le stockage en mémoire sera effectué par le runtime, et que l'IP gérera lui-même la récupération des données d'entrée et le stockage des données de sortie. Dans le cas matériel, les descripteurs de contrôle et de ports sont communs à l'encodeur et au décodeur, en raison de leur interface similaire.

5.3.2 Tests menés et résultats

Nous présentons dans cette section les configurations utilisées pour les tests, puis les résultats de ceux-ci.

```
<Implementation_port_descriptor ID="17">
  <Parameter Name="Input_data">
    <Set>
      <Action Type="Pointer" Direction="Write"
        Data_parameter="" Count="0x00028000" Size="16">
        <Address Offset="24" Size="4"/>
        <Count Offset="32" Size="4"/>
      </Action>
    </Set>
  </Parameter>
  <Return_value Name="Output_data">
    <Get>
      <Action Type="Pointer" Direction="Read"
        Data_parameter="" Count="0x00028000" Size="16">
        <Address Offset="28" Size="4"/>
      </Action>
    </Get>
    <Return_value_ready_hint>
      <Action Type="Register" Direction="Read"
        Hint="" Offset="8" Count="1" Size="4">
        <Neq Value="0"/>
      </Action>
    </Return_value_ready_hint>
  </Return_value>
</Implementation_port_descriptor>
```

Figure 5.19 – *Exemple de fichier de données ordonnées.*

5.3.2.a Configuration des tests

L'exécution utilisant les noyaux matériels présente deux phases : la reconfiguration puis l'exécution elle-même. Pour que l'exécution matérielle soit intéressante, il faut que le temps de reconfiguration soit faible, voire négligeable face au temps d'exécution. Afin de vérifier ce point, on exécute l'application de tests en utilisant plusieurs tailles de données.

Pour que le test soit représentatif du cryptage d'un fichier, nous avons choisi des quantités de données de 512 Kio, 1 Mio et 5 Mio. Concernant les noyaux matériels, nous avons suffisamment de place sur le dispositif pour placer deux cellules de taille suffisante pour les noyaux AES. Sachant que les deux noyaux doivent s'exécuter l'un après l'autre, l'application n'a besoin que d'une seule cellule. Grâce à la présence de ces deux cellules, nous avons la possibilité de séparer le flux de données en deux, et d'exécuter simultanément deux encodeurs, puis deux décodeurs afin d'accélérer le traitement. On présente cette variante de l'application sur la figure 5.20.

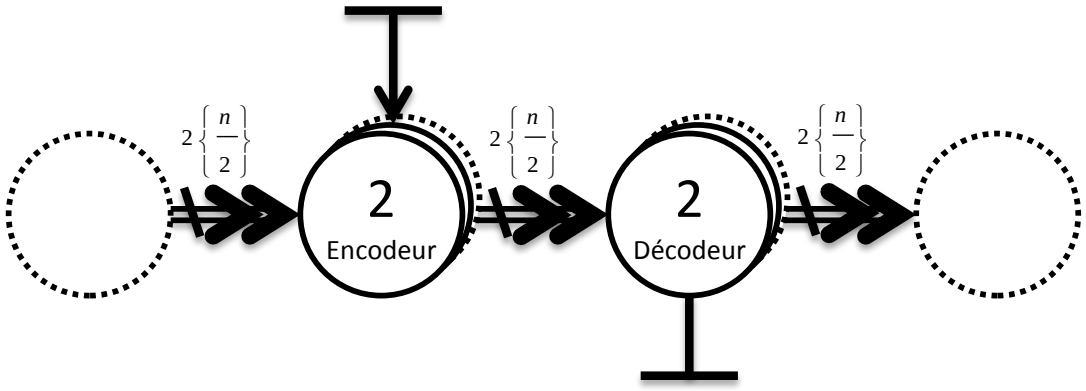


Figure 5.20 – *Modélisation de l'application de test utilisant deux branches parallèles.*

Pour cela, il nous suffit de déclarer deux implémentations matérielles pour chaque noyau, correspondant chacune à une cellule, et de générer les fichiers de configuration correspondant pour chaque noyau. Par ailleurs, le descripteur d'application sera modifié de manière à dupliquer les appels aux noyaux, chacun opérant sur la moitié des données.

Ainsi, nous exécuterons l'application en utilisant les noyaux logiciels, les noyaux matériels sur une cellule, et les noyaux matériels sur deux cellules. Chacun de ces tests sera réalisé sur les trois quantités de données.

5.3.2.b Résultats

Chaque test a été exécuté 10 fois, et les résultats recensés dans le tableau 5.3 présentent les temps moyens des réalisations, ainsi que la marge d'erreur correspondant à la variation rencontrée lors des tests. Les gains de l'exécution à l'aide de noyaux matériels par rapport aux noyaux logiciels sont présentés dans le tableau 5.4.

La principale observation que l'on peut faire sur ces résultats est que l'utilisation d'accélérateurs matériels, même devant subir une phase de reconfiguration, est plus intéressante que l'exécution logicielle sur processeurs PowerPC. Dans le cadre de l'embarqué, on aura donc intérêt à privilégier des noyaux matériels reconfigurables par rapport à l'exécution logicielle lorsque celle-ci est disponible.

Dans le cas de figure où l'exécution matérielle est distribuée sur deux cellules, le temps de reconfiguration augmente, car la reconfiguration est séquentielle, mais nous constatons néanmoins que le gain global par rapport au logiciel est encore plus grand qu'en utilisant une seule cellule.

5.4 Conclusion sur la plateforme HSDP

L'implémentation de la reconfiguration matérielle sur la plateforme SPoRE nous a permis de valider le modèle SPoRE. En effet, il a été possible d'implémenter des noyaux reconfigurables sur la base de la réutilisation d'IPs existants. Par ailleurs, nous avons pu constater que le temps de reconfiguration des noyaux pouvait être négligeable en comparaison du gain réalisé par rapport à une exécution logicielle. Nous avons également validé le modèle décentralisé de SPoRE, avec récupération des composants de l'application sur un serveur de données.

Après une première plateforme MPI, la plateforme HSDP confirme donc la pertinence de l'architecture SPoRE en démontrant la facilité d'utilisation de la RDP. De plus, l'utilisation de noyaux hétérogènes, fournissant à la fois une implémentation logicielle et une implémentation matérielle, nous a permis de valider notre couche de virtualisation et le principe de nos accesseurs. En effet, la même application, travaillant sur les mêmes fichiers de données, a pu fonctionner à la fois en logiciel et en matériel simplement en fournissant les deux implémentations associées aux descripteurs adéquats.

Bien que cette plateforme ne reprenne pas l'intégralité des spécifications de SPoRE, la pertinence des briques essentielles se voit donc confirmée. Néanmoins, un travail reste à fournir concernant l'ordonnancement. Ainsi, l'ordonnanceur global devra pouvoir répartir les noyaux sur les nœuds en fonction de critères définis par l'utilisateur.

Quantité de données traitées			512 Kio	1 Mio	5 Mio
Nature des noyaux	Noyau	Durée ...			
Logiciels	Encodeur	d'exécution	426 580 $\mu s \pm$ 0,37%	717 269 $\mu s \pm$ 0,63%	3 049 773 $\mu s \pm$ 0,83%
	Décodeur	d'exécution	550 588 $\mu s \pm$ 0,87%	972 432 $\mu s \pm$ 0,23%	4 349 304 $\mu s \pm$ 0,63%
Matériels (1 cellule)	Encodeur	d'exécution	52 926 $\mu s \pm$ 0,03%	105 847 $\mu s \pm$ 0,02%	529 166 $\mu s \pm$ 0,01%
	Décodeur	d'exécution	52 942 $\mu s \pm$ 0,06%	105 853 $\mu s \pm$ 0,01%	529 178 $\mu s \pm$ 0,01%
	Encodeur	de reconfiguration	825 $\mu s \pm$ 0,67%		
	Décodeur	de reconfiguration	906 $\mu s \pm$ 0,12%		
	Encodeur	totale	53 751 $\mu s \pm$ 0,03%	106 673 $\mu s \pm$ 0,02%	529 991 $\mu s \pm$ 0,01%
	Décodeur	totale	53 848 $\mu s \pm$ 0,06%	106 759 $\mu s \pm$ 0,01%	530 084 $\mu s \pm$ 0,01%
Matériels (2 cellules)	Encodeur	d'exécution	28,541 $\mu s \pm$ 0.15%	57,974 $\mu s \pm$ 0.08%	285,656 $\mu s \pm$ 0.08%
	Décodeur	d'exécution	26,859 $\mu s \pm$ 0.03%	54,092 $\mu s \pm$ 0.04%	271,887 $\mu s \pm$ 0.05%
	Encodeur	de reconfiguration	1,671 $\mu s \pm$ 0.46%		
	Décodeur	de reconfiguration	1,820 $\mu s \pm$ 0.21%		
	Encodeur	totale	30,177 $\mu s \pm$ 0.15%	59,642 $\mu s \pm$ 0.09%	287,329 $\mu s \pm$ 0.08%
	Décodeur	totale	28,670 $\mu s \pm$ 0.03%	55,912 $\mu s \pm$ 0.04%	273,706 $\mu s \pm$ 0.05%

Tableau 5.3 – *Temps d'exécution de chaque noyau de l'application AES selon les différentes configurations.*

Quantité de données traitées		512 Kio	1 Mio	5 Mio
Une cellule	Gain par rapport au logiciel	88,99%	87,37%	85,67%
	Durée de reconfiguration	1,64%	0,82%	0,16%
Deux cellules	Gain par rapport au logiciel	93,98%	93,16%	92,42%
	Durée de reconfiguration	6,3%	3,12%	0,63%

Tableau 5.4 – *Comparaison des temps d'exécution.*

Chapitre 6

Conclusion

Sommaire

Dans ce document, j'ai présenté mes travaux, consistant en un flot complet de conception et d'exécution des applications parallèles hétérogènes. Ce flot est constitué d'un ensemble d'outils pour la description et la prise en charge de ces applications. J'ai ainsi détaillé le modèle d'applications, comprenant une représentation graphique des applications découpées en noyaux de calcul, et séparant le contrôle. Par la suite, j'ai présenté la méthodologie pour implémenter les applications, selon une correspondance directe avec le modèle, qui permet de décrire la couche contrôle de l'application, et propose de combiner plusieurs implémentations pour chaque noyau, afin de permettre l'exécution sur différentes ressources de calcul. Enfin, j'ai détaillé SPoRE, la plateforme d'exécution théorique pour le déploiement d'applications, ainsi que deux implémentations de celle-ci. Dans ce chapitre, il s'agira de faire le bilan de cet ensemble, en explorant notamment les éléments restant à développer, et les différents travaux futurs en résultant.

6.1 Bilan

Dans le chapitre 1, j'ai présenté les motivations ayant conduit à mener ces travaux. J'y ai identifié les objectifs à mener à bien, et défini un ensemble d'éléments à mettre en place pour atteindre ces objectifs.

Puis, dans le chapitre 2, j'ai effectué un tour d'horizon de l'état de l'art dans les domaines qui nous intéressent. Ainsi, j'ai défini les termes importants, formés par les couples logiciel/matériel et application/plateforme. Sur cette base, nous nous sommes intéressés au domaine du parallélisme, dont nous avons vu les différents niveaux existants. Par la suite, nous avons étudié les avantages de l'utilisation d'éléments de calcul matériel. À ce propos, nous nous sommes particulièrement attardés sur le principe de la reconfiguration matérielle et les systèmes qui les utilisent, tels les HPRCs. Puis, nous avons recensé différentes natures d'interfaces, moyens de communication entre les IPs. Finalement, nous avons détaillé quelques plateformes existantes prenant en charge la reconfiguration matérielle, ainsi que des travaux concernant l'ordonnancement.

Dans le chapitre 3, j'ai spécifié l'ensemble des outils que je me proposais de développer pour répondre à la problématique. Ainsi, j'ai défini un modèle d'application parallèle, séparant le contrôle du traitement afin de permettre différents types de noyaux de calcul. Ce modèle a été assorti d'une représentation graphique afin de permettre de décrire aisément ces applications. J'ai détaillé les relations possibles entre les noyaux de calculs, ainsi que les structures de contrôles telles que les boucles et les tests conditionnels. Concernant la représentation des noyaux, j'ai défini une couche de virtualisation autorisant au développeur la soumission de différentes implémentations pour un même noyau, afin de supporter plusieurs natures d'unités d'exécution. En sus de ce modèle, j'ai proposé une plateforme, que j'ai appelé SPoRE, et qui permet l'exécution d'applications définies à l'aide de ce

modèle. Cette plateforme supporte des unités d'exécution logicielles et matérielles, tant statiques que dynamiques. Il s'agit d'une plateforme à architecture distribuée, inspirée du modèle des HPCs, et supportant une structure hétérogène. Celle-ci propose différentes unités d'exécution, les *cellules*, rassemblées en *nœuds* dialoguant via un réseau. Chaque nœud dispose d'une cellule hôte, en charge de la gestion, et des cellules de calcul hébergeant les noyaux.

À la suite de ces définitions, j'ai proposé deux implémentations de la plateforme SPoRE, détaillées dans les chapitres 4 et 5.

Dans le chapitre 4, j'ai présenté une plateforme utilisant des PEs logiciels, dont le code peut-être reconfiguré afin de permettre l'exécution de différentes applications. Cette implémentation est basée sur MPI, une bibliothèque permettant l'échange de messages entre les noyaux d'une application parallèle. Le support de MPI permet une compatibilité avec la plupart des applications à mémoire distribuée, majoritairement basées sur cette spécification. Sur cette base, j'ai défini un protocole de délégation des noyaux de calculs sur les cellules de calcul en reconfigurant le code logiciel exécuté par celle-ci. Enfin, j'ai testé cette plateforme à l'aide d'un benchmark pour HPC, que j'ai adapté à la plateforme. J'ai ainsi mis en lumière certaines faiblesses de la plateforme, notamment en ce qui concerne l'accès simultané à la mémoire par les différentes cellules, formant un goulet d'étranglement et impactant les performances.

La seconde implémentation de SPoRE, présentée dans le chapitre 5, est quant à elle orientée vers les applications flot de données et le support de la reconfiguration dynamique partielle. Pour celle-ci, j'ai adopté une approche différente de la première implémentation tout en conservant un certain nombre d'éléments communs. La structure de base a été conservée, avec un réseau rassemblant des nœuds formés de cellules, et la hiérarchie entre la cellule hôte et les cellules de calcul a été maintenue. Néanmoins, cette plateforme utilise cette fois-ci des cellules reconfigurables, permettant d'intégrer des PEs par reconfiguration partielle. La cellule hôte a cette fois-ci un rôle étendu, avec un runtime prenant en charge la totalité du flot d'exécution, et notamment l'ordonnancement, qui était dévolu à MPI dans l'implémentation précédente. Cette plate-forme supporte l'exécution d'applications décrites en XML selon une syntaxe que nous avons définie.

Cet ensemble d'outils supporte les différentes étapes de la conception et de l'exécution d'applications parallèles hétérogènes. De plus, en ajoutant la gestion automatique de la reconfiguration dynamique partielle, ceci permet de délester le développeur de la gestion de celle-ci, tâche relativement lourde à l'heure actuelle. Bien que certains éléments manquent encore (voir section suivante sur les travaux futurs), je suis arrivé à définir et mettre en œuvre un flot complet, qui a pu être testé avec succès. Ce flot, représenté schématiquement sur la figure 6.1 permet différentes utilisations. En effet, le déploiement automatisé d'applications sur plateforme distribuée permet d'envisager des usages autres que le déploiement final

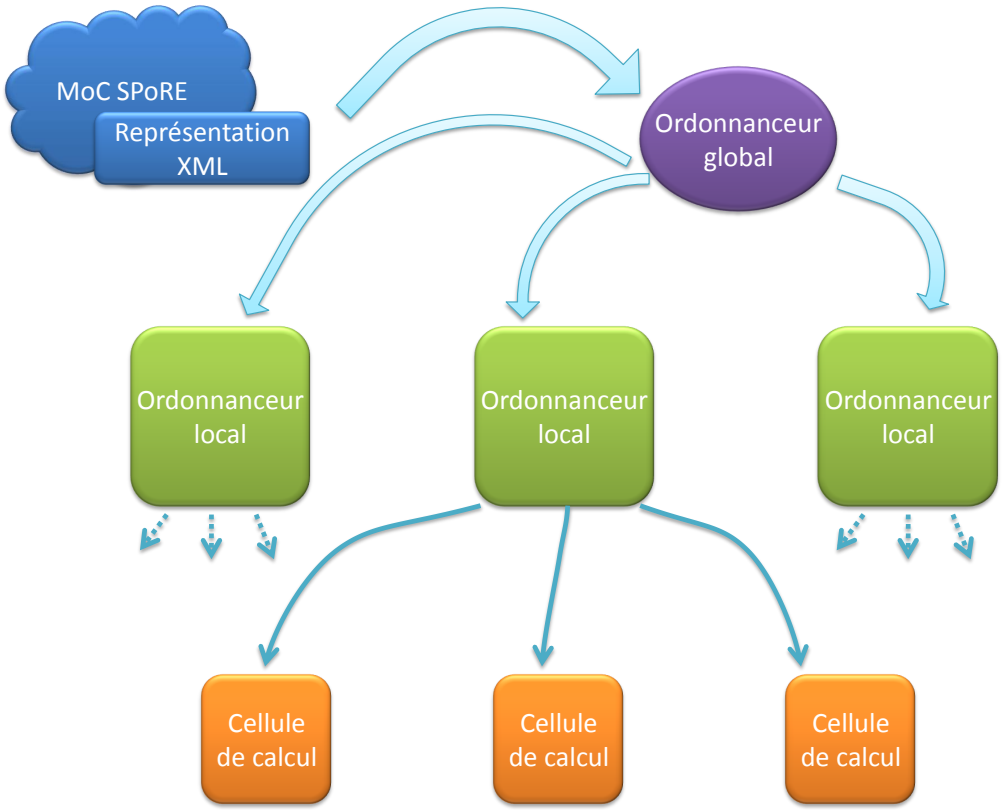


Figure 6.1 – *Représentation fonctionnelle de la plateforme SPoRE.*

d'applications. Ainsi, pour réaliser le test d'un noyau, la description rapide de l'application en XML et le déploiement automatisé permettent d'obtenir un résultat dans des délais très courts. De plus, le support du parallélisme de nœud permet d'exécuter simultanément plusieurs configurations d'un même noyau.

La modularité des services composant le runtime SPoRE permet également très simplement à un utilisateur de changer certains comportements de la plateforme. Ainsi, pour modifier la politique d'ordonnancement, il suffira de décrire le service correspondant, et de l'utiliser en lieu et place de celui fourni avec la plateforme. De plus, les deux niveaux d'ordonnancement peuvent être modifiés indépendamment l'un de l'autre, permettant encore plus de modularité.

Durant cette thèse, plusieurs articles ont été publiés dans des conférences, ainsi que dans une revue. Le premier article [30] m'a permis de présenter les travaux concernant la première implémentation de SPoRE. Par la suite, dans l'article [31], j'ai présenté le flot de conception plus en détail, ainsi qu'une première approche de la deuxième plateforme. Puis, la revue [33] m'a permis de détailler plus en détail la plateforme théorique SPoRE, ainsi que les deux implémentations de celle-ci. Enfin, dans l'article [32], je présente en particulier le mécanisme de gestion automatique des interfaces pour les IPs matériels.

6.2 Travaux futurs et perspectives

L'aspect principal manquant dans SPoRE concerne l'ordonnancement. En effet, si nous avons pu réaliser de l'ordonnancement multi-nœud, c'est en utilisant un ordonnanceur MPI déjà existant, destiné au logiciel homogène. Or, l'intérêt même de la plateforme SPoRE est de proposer une architecture hétérogène, associée à de multiples implémentations des noyaux. Dans ces conditions, le choix de l'implémentation est un élément important, et il convient de développer l'aspect ordonnancement.

Nous avons déjà commencé à identifier les paramètres importants pour l'ordonnancement, en termes de performance, que ce soit par rapport au choix de l'implémentation ou par rapport à la localisation dans le réseau SPoRE, en fonction de la vitesse des liens. Néanmoins, n'ayant pas encore implémenté ceux-ci, nous n'avons pas pu vérifier leur pertinence, et préférons donc considérer ceci comme des travaux futurs.

On pourrait également envisager différentes métriques, liées à l'aspect auto-adaptatif. Par exemple, outre les critères de performance, il peut être utile de prendre en compte l'aspect consommation, prépondérant dans les systèmes embarqués. Ainsi, on pourrait imaginer, lors de l'ordonnancement d'une boucle sur un noyau, de changer l'implémentation utilisée pour le noyau d'une itération à l'autre, car la batterie est passée sous un seuil d'alerte. De la même manière, la définition d'un profil temps-réel serait tout à fait pertinente pour certaines applications, bien que dans ce cas là, l'indépendance vis à vis de la plateforme ne soit plus garantie.

De plus, les différentes techniques d'ordonnancement que nous avons étudiées dans l'état de l'art offrent des pistes intéressantes. Ainsi, Application Heartbeats, couplé à la couche de virtualisation que nous avons proposée, permettrait de s'affranchir du programme de conversion des données entre implémentation. En effet, notre couche de virtualisation est forcément fournie avec chaque noyau, et permet de stocker les données sous une forme indépendante de l'implémentation. Le choix dynamique de l'implémentation offert par Application Heartbeats pourrait par exemple être envisagé pour l'exécution de boucles sur un même noyau. AMAP offre également une approche adaptative intéressante : en fonction des exécutions passées d'un noyau et de la taille des paramètres correspondant à celles-ci, cette technique permet de définir quelle implémentation sera la plus intéressante lors des exécutions futures. Tous ces éléments peuvent former des heuristiques intéressantes qu'il faudra prendre en compte lors du choix des métriques utilisées pour l'ordonnancement.

Concernant l'aspect implémentation, un autre élément restant à réaliser est une plateforme qui unifierait l'aspect MPI avec la reconfiguration matérielle. Initialement envisagée en tant que troisième implémentation de SPoRE, celle-ci pourrait néanmoins être intégrée à la seconde, en ajoutant le support MPI en même temps

que l'intégration des différents composants d'ordonnancement.

Un autre point concerne le passage à l'échelle des nœuds. En effet, la structure inter-nœuds de SPoRE est conçue pour un grand nombre de ressources, avec de multiples serveurs de données et peu d'échanges avec l'ordonnanceur global. En revanche, comme on a pu le constater sur la première implémentation, l'architecture du nœud peut mener à un engorgement de l'accès à la mémoire, d'autant plus si le nombre de PEs augmente. Ainsi, une piste de travail est l'utilisation des NoCs, chaque PE disposant de sa mémoire locale sur laquelle il peut travailler sans conflit. Par ailleurs, l'utilisation de bus du type AXI stream, permettant des transferts en flot de données, offre des pistes intéressantes pour les échanges de données entre cellule hôte et cellule de calcul.

Enfin, un élément qui serait envisageable à terme concerne la génération automatique des fichiers de configuration du FPGA à l'exécution. En effet, actuellement, les fichiers de configuration doivent être synthétisés préalablement par l'utilisateur, qui doit ainsi envisager les différentes possibilités de placement du noyau. La génération dynamique des fichiers de configuration permettrait de ne générer que les fichiers de configuration réellement utilisés. En contrepartie, cette génération n'étant pas instantanée, cela rajouterait un temps de latence supplémentaire avant l'exécution. Pour certains cas néanmoins, la recherche de la vitesse d'exécution est moins importante que l'optimisation de la capacité de stockage, dont l'usage peut-être réduit par ce moyen.

Bibliographie

[1] ARM Limited, `http://www.arm.com/products/system-ip/amba/amba-open-specifications.php`. *AMBA Specification (Rev 2.0)*, 1999.

[2] ARM Limited, `http://www.arm.com/products/system-ip/amba/amba-open-specifications.php`. *MBA AXI Protocol Version : 2.0 Specification*, 2010.

[3] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research : A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf`, december 2006.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical report, NASA Advanced Supercomputing, `http://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf`, 1994.

[5] Giovanni Beltrame, Luca Fossati, and Donatella Sciuto. High-level modeling and exploration of reconfigurable MPSoCs. In *Proceedings of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, AHS '08, pages 330–337, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Luca Benini and Giovanni De Micheli. Networks on chips : A new SoC paradigm. *Computer*, 35 :70–78, 2002.

[7] Neil Bergmann, John Williams, and Peter Waldeck. Egret : A flexible platform for real-time reconfigurable systems on chip. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 300–303, Las Vegas, USA, 2003.

[8] Yitzhak Birk and Evgeny Fiksman. Dynamic reconfiguration architectures for multi-context FPGAs. *Comput. Electr. eng.*, 35 :878–903, November 2009.

[9] Mark Burton. *The OCP IP-core interface Standard : Supporting ESL*. OCP-IP, Electronica, 2006.

[10] Jose Antonio Casas, Juan Manuel Moreno, Jordi Madrenas, and Joan Cabestany. A novel hardware architecture for self-adaptive systems. In *AHS '07 : Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 592–599, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Lawrence Snyder, W. Derrick Weathersby, and Calvin Lin. The case for high-level parallel programming in ZPL. *IEEE Comput. Sci. Eng.*, 5 :76–86, July 1998.

[12] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[13] Guillaume Colin de Verdière. Introduction to GPGPU, a hardware and software background. *Comptes Rendus Mécanique*, 339(2–3) :78 – 89, 2011.

[14] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP : deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM.

[15] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical report, Knoxville, TN, USA, 1989.

[16] Jack J. Dongarra. Performance of various computers using standard linear equations software. *SIGARCH Comput. Archit. News*, 18(1) :17–, March 1990.

[17] Jack J. Dongarra. Performance of various computers using standard linear equations software. *SIGARCH Comput. Archit. News*, 20(3) :22–44, June 1992.

[18] Jack J. Dongarra. Performance of various computers using standard linear equations software. *University of Tennessee*, 2008.

[19] Jack J. Dongarra. Performance of various computers using standard linear equations software. *University of Tennessee*, 2010.

[20] Jack J. Dongarra. Performance of various computers using standard linear equations software. *University of Tennessee*, 2011.

[21] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark : past, present and future. *Concurrency Comput. Pract. Ex.*, 15(9) :803–820, 2003.

[22] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL : Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8) :391 – 407, 2012.

[23] François Duhem, Fabrice Muller, and Philippe Lorenzini. FaRM : Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA. In Andreas Koch, Ram Krishnamurthy, John McAllister, Roger Woods, and Tarek El-Ghazawi, editors, *Reconfigurable Computing : Architectures, Tools and Applications*, volume 6578 of *Lecture Notes in Computer Science*, pages 253–260. Springer Berlin / Heidelberg, 2011.

[24] François Duhem, Fabrice Muller, and Philippe Lorenzini. Reconfiguration time overhead on field programmable gate arrays : reduction and cost model. *IET Computers & Digital Techniques*, 6(2) :105 – 113, March 2012.

[25] Esam El-Araby, Ivan Gonzalez, and Tarek El-Ghazawi. Exploiting partial runtime reconfiguration for high-performance reconfigurable computing. *ACM Trans. Reconfigurable Technol. Syst.*, 1 :21 :1–21 :23, January 2009.

[26] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The promise of high-performance reconfigurable computing. *Computer*, 41 :69–76, February 2008.

[27] Famille ARM Cortex. `http://http://www.arm.com/products/processors/cortex-a/index.php`, 2012.

[28] Roger Ferrer, Judit Planas, Pieter Bellens, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Rosa Badia, Eduard Ayguade, and Jesus Labarta. Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL. In Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, editors, *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin / Heidelberg, 2011.

[29] Ehud Finkelstein and Shlomo Weiss. Microprocessor system buses : A case study. *Journal of Systems Architecture*, 45(12–13) :1151 – 1168, 1999.

[30] Clément Foucher, Fabrice Muller, and Alain Giulieri. Exploring FPGAs capability to host a HPC design. In *28th Norchip Conference (Norchip 2010)*, pages 1–4, Tampere, Finland, November 2010.

[31] Clément Foucher, Fabrice Muller, and Alain Giulieri. Flot de conception d'applications parallèles sur plateforme reconfigurable dynamiquement. In *14e Symposium en Architectures nouvelles de machines (SympA 2011)*, 2011.

[32] Clément Foucher, Fabrice Muller, and Alain Giulieri. Fast integration of hardware accelerators for dynamically reconfigurable architecture. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2012)*, York, U.K., July 2012.

[33] Clément Foucher, Fabrice Muller, and Alain Giulieri. Online codesign on reconfigurable platform for parallel computing. *Microprocess. Microsyst.*, 2012.

[34] Edgar Gabriel, Graham Fagg, George Bosilca, Thara Angskun, Jack Dongarra, Jeffrey Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph Castain, David Daniel, Richard Graham, and Timothy Woodall. Open MPI : Goals, concept, and design of a next generation MPI implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 353–377. Springer Berlin / Heidelberg, 2004.

[35] Shanyuan Gao, Andrew G. Schmidt, and Ron Sass. Impact of reconfigurable hardware on accelerating mpi_reduce. In Jinian Bian, Qiang Zhou, Peter Athanas, Yajun Ha, and Kang Zhao, editors, *FPT*, pages 29–36. IEEE, 2010.

[36] Akila Gothandaraman, Gregory D. Peterson, G. L. Warren, Robert J. Hinde, and Robert J. Harrison. FPGA acceleration of a quantum monte carlo application. *Parallel Comput.*, 34 :278–291, May 2008.

[37] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.*, 22 :789–828, September 1996.

[38] Hans Hacker, Carsten Trinitis, Josef Weidendorfer, and Matthias Brehm. Considering GPGPU for HPC centers : Is it worth the effort ? In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore-Challenge*, volume 6310 of *Lecture Notes in Computer Science*, pages 118–130. Springer Berlin / Heidelberg, 2011.

[39] Georg Hager and Gerhard Wellein. Architecture and performance characteristics of modern high performance computers. In H. Fehske, R. Schneider, and A. Weiße, editors, *Computational Many-Particle Physics*, volume 739 of *Lecture Notes in Physics*, pages 681–730. Springer Berlin / Heidelberg, 2008.

[40] Rolf Hempel and David W. Walker. The emergence of the MPI message passing standard for parallel computing. *Comput. Stand. Interfaces*, 21 :51–62, May 1999.

[41] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7) :33–38, July 2008.

[42] Jay P. Hoeflinger and Bronis R. De Supinski. The OpenMP memory model. In *Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming*, IWOMP'05/IWOMP'06, pages 167–177, Berlin, Heidelberg, 2008. Springer-Verlag.

[43] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA '05 : Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[44] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, and et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. *Dig. Tech. Pap. IEEE Int. Solid. State. Circuits Conf.*, February 2010.

[45] IBM, `https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture`. *128-Bit Processor Local Bus Architecture Specifications Version 4.7*, 2007.

[46] R. A. Iushchenko. Measuring the performance of parallel computers with distributed memory. *Cybernetics and Sys. Anal.*, 45 :941–951, November 2009.

[47] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang, and Barbara Chapman. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*, 37(9) :562 – 575, 2011.

[48] Jaume Joven, Oriol Font-Bach, David Castells-Rufas, Ricardo Martinez, Lluis Teres, and Jordi Carrabina. xENoC - an experimental network-on-chip environment for parallel distributed computing on NoC-based MPSoC architectures. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0 :141–148, 2008.

[49] Abdullah Kayi, Tarek El-Ghazawi, and Gregory B. Newby. Performance issues in emerging homogeneous multi-core architectures. *Simulation Modelling Practice and Theory*, 17(9) :1485 – 1499, 2009.

[50] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[51] Srinidhi Kestur, John D. Davis, and Oliver Williams. BLAS comparison on FPGA, CPU and GPU. In *Proceedings of the 2010 IEEE Annual Symposium on VLSI*, ISVLSI '10, pages 288–293, Washington, DC, USA, 2010. IEEE Computer Society.

[52] P. Kwan and C. Clarke. FPGAs for improved energy efficiency in processor based systems. In Thambipillai Srikanthan, Jingling Xue, and Chip-Hong Chang, editors, *Advances in Computer Systems Architecture*, volume 3740 of *Lecture Notes in Computer Science*, pages 440–449. Springer Berlin / Heidelberg, 2005.

[53] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Run-time support for heterogeneous multitasking on reconfigurable SoCs. *Integrat. VLSI J.*, 38(1) :107 – 130, 2004.

[54] SPIRIT Schema Working Group Membership. *IP-XACT User Guide v1.2*. `www.spiritconsortium.org`, July 2006.

[55] Michael Metcalf and John Reid. *Fortran 90 explained*. Oxford University Press, Inc., New York, NY, USA, 1990.

[56] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 16 :1–16 :14, New York, NY, USA, 2008. ACM.

[57] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2) :1–31, August 1998.

[58] OCP-IP, `http://www.ocpip.org`. *Open Core Protocole Specification*, 2001.

[59] Resources open source Xilinx. Xilinx, Inc., `http://wiki.xilinx.com`, 2012.

[60] OpenCores, bilbiothèque d'IPs libres et gratuits. `http://opencores.org`, 2012.

[61] Yoshio Oyanagi. Future of supercomputing. *J. Comput. Appl. Math.*, 149(1) :147–153, 2002.

[62] John Reid. The new features of fortran 2003. *SIGPLAN Fortran Forum*, 26(1) :10–33, April 2007.

[63] Mehran Rezaei and Krishna M. Kavi. Intelligent memory manager : Reducing cache pollution due to memory management functions. *Journal of Systems Architecture*, 52(1) :41 – 55, 2006.

[64] Harvey Richardson. High performance fortran : history, overview and current developments. Technical report, 1.4 TMC-261, Thinking Machines Corporation, 1996.

[65] Silcore, Opencores.org. *Specification for the WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, 2002.

[66] Vlad-Mihai Sima and Koen Bertels. Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.

[67] F. Sironi, M. Triverio, H. Hoffmann, M. Maggio, and M.D. Santambrogio. Self-aware adaptation in FPGA-based systems. *International Conference on Field Programmable Logic and Applications*, 0 :187–192, 2010.

[68] Filippo Sironi and Andrea Cuoccio. Self-aware adaptation via implementation hot-swap for heterogeneous computing. In *Proceedings of the 2011 1st International Workshop on Computing in Heterogeneous, Autonomous 'N' Goal-Oriented Environments*, CHANGE '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[69] Site du classement des HPCs TOP500. `http://top500.org`, 2012.

[70] Site du NAS Parallel Benchmark. `http://www.nas.nasa.gov/publications/npb.html`, 2012.

[71] Site du projet Æther. `http://www.aether-ist.org`, 2006.

[72] Site officiel d'OpenCL. `http://www.khronos.org/opencl`, 2012.

[73] Site web Adobe Flash Player. `http://www.adobe.com/fr/products/flashplayer.html`, 2012.

[74] Site web de BusyBox. `http://www.busybox.net`, 2012.

[75] Site web de l'application VLC. `http://www.videolan.org/vlc`, 2012.

[76] Site web de μClibc. `http://uclibc.org`, 2012.

[77] Site web du projet Buildroot. `http://buildroot.uclibc.org`, 2012.

[78] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30 :123–169, June 1998.

[79] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Trans. Embed. Comput. Syst.*, 7 :14 :1–14 :28, January 2008.

[80] Erich Strohmaier, Jack J. Dongarra, Hans W. Meuer, and Horst D. Simon. The marketplace of high-performance computing. *Parallel Comput.*, 25 :1517–1544, December 1999.

[81] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[82] Xian-He Sun and Yong Chen. Reevaluating Amdahl's law in the multicore era. *J. Parallel Distrib. Comput.*, 70 :183–188, February 2010.

[83] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic OpenCL device characterization : Guiding optimized kernel design. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 438–452. Springer Berlin / Heidelberg, 2011.

[84] Matthew A. Watkins and David H. Albonesi. ReMAP : A reconfigurable heterogeneous multicore architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 497–508, Washington, DC, USA, 2010. IEEE Computer Society.

[85] David A. Wheeler. Ada, C, C++, and Java vs. the Steelman. *Ada Lett.*, XVII(4) :88–112, July 1997.

[86] John Williams and Neil Bergmann. Embedded linux as a platform for dynamically self-reconfiguring systems-on-chip. In *The International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 163–169, Las Vegas, Nevada, USA, June 2004. CSREA Press.

[87] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov. OpenFPGA CoreLib core library interoperability effort. *Parallel Comput.*, 34(4-5) :231 – 244, 2008. Reconfigurable Systems Summer Institute 2007.

[88] Dong Hyuk Woo and Hsien-Hsin S. Lee. Extending Amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41 :24–31, 2008.

[89] Xilinx, Inc. *Partial Reconfiguration User Guide*. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/ug702.pdf`, October 2010.

[90] Xilinx, Inc. *Microblaze processor reference guide*. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/mb_ref_guide.pdf`, June 2011.

[91] Linfeng Ye, Jean-Philippe Diguet, and Guy Gogniat. Reconfigurable MPSoCs for on-demand computing. In *Reconfigurable MPSoCs for On-Demand Computing*, page 1, Dijon, France, September 2009.

[92] Won young Chung, Ha young Jeong, Won Woo Ro, and Yong-Surk Lee. A low-cost standard mode mpi hardware unit for embedded mpsoc. *IEICE Transactions*, 94-D(7) :1497–1501, 2011.